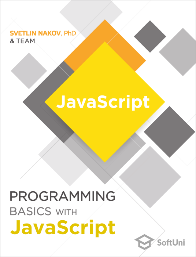
# Free Book "Programming Basics with JavaScript"

The free book "**Programming Basics with JavaScript**" introduces the readers to writing **programming code** at a beginner level (variables and data, conditional statements, loops, and functions) using the **JavaScript** programming language.

[](https://js-book.softuni.org/)

* [Preface](https://js-book.softuni.org/chapter-00-preface.html)
* [1. First Steps in Programming](https://js-book.softuni.org/chapter-01-first-steps-in-programming.html)
* [2.1. Simple Calculations](https://js-book.softuni.org/chapter-02-simple-calculations.html)
* [2.2. Simple Calculations – Exam Problems](https://js-book.softuni.org/chapter-02-simple-calculations-exam-problems.html#simple-calculations)
* [3.1. Simple Conditions](https://js-book.softuni.org/chapter-03-simple-conditions.html)
* [3.2. Simple Conditions – Exam Problems](https://js-book.softuni.org/chapter-03-simple-conditions-exam-problems.html)
* [4.1. More Complex Conditions](https://js-book.softuni.org/chapter-04-complex-conditions.html)
* [4.2. More Complex Conditions – Exam Problems](https://js-book.softuni.org/chapter-04-complex-conditions-exam-problems.html)
* [5.1. Loops](https://js-book.softuni.org/chapter-05-loops.html)
* [5.2. Loops – Exam Problems](https://js-book.softuni.org/chapter-05-loops-exam-problems.html)
* [6.1. Nested Loops](https://js-book.softuni.org/chapter-06-nested-loops.html)
* [6.2. Nested Loops – Exam Problems](https://js-book.softuni.org/chapter-06-nested-loops-exam-problems.html)
* [7.1. More Complex Loops](https://js-book.softuni.org/chapter-07-complex-loops.html)
* [7.2. More Complex Loops – Exam Problems](https://js-book.softuni.org/chapter-07-complex-loops-exam-problems.html)
* [8.1. Practical Exam Preparations – Part I](https://js-book.softuni.org/chapter-08-exam-preparation.html)
* [8.2. Practical Exam Preparations – Part II](https://js-book.softuni.org/chapter-08-exam-preparation-part-2.html)
* [9.1. Problems for Champions – Part I](https://js-book.softuni.org/chapter-09-problems-for-champions.html)
* [9.2. Problems for Champions – Part II](https://js-book.softuni.org/chapter-09-problems-for-champions-part-2.html)
* [10. Functions](https://js-book.softuni.org/chapter-10-functions.html)
* [11. Tricks and Hacks](https://js-book.softuni.org/chapter-11-tricks-and-hacks.html)
* [Conclusion](https://js-book.softuni.org/chapter-12-conclusion.html)

## Download The Book

Download the book "**Programming Basics with JavaScript**" in **PDF** format:

* [TODO](https://js-book.softuni.org/)

## Book Editions

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

* [Programming Basics with C# (English)](https://csharp-book.softuni.org/)
* [Programming Basics with Java (English)](https://java-book.softuni.org/)
* [Programming Basics with JavaScript (English)](https://js-book.softuni.org/)
* [Programming Basics with Python (English)](https://python-book.softuni.org/)
* [Programming Basics with C# (Bulgarian)](https://csharp-book.softuni.bg/)
* [Programming Basics with Java (Bulgarian)](https://java-book.softuni.bg/)
* [Programming Basics with JavaScript (Bulgarian)](https://js-book.softuni.bg/)
* [Programming Basics with Python (Bulgarian)](https://python-book.softuni.bg/)
* [Programming Basics with C++ (Bulgarian)](https://cpp-book.softuni.bg/)

## Book Details

Brief information about this edition:

* Title: **Programming Basics with JavaScript**
* Authors: **Svetlin Nakov & Team**
  + Main authors: Boncho Vulkov, Christian Hristov, Dimitar Dalev, Elena Rogleva, Hristo Minkov, Ivelin Arnaudov, Julieta Atanasova, Kristian Marianov, Martin Chaov, Nikolay Bankin, Nikolay Kostov, Pavel Kolev, Petar Ivanov, Stiliyan Kangalov, Svetlin Nakov, Ventsislav Petrov, Zahariya Pehlyavanova, Zdravko Kostadinov
  + Other contributors: Aleksander Krastev, Aleksander Lazarov, Aleksander Peev, Alen Paunov, Angel Dimitriev, Ariet Motzeva, Daniel Tsvetkov, Denis Milanov, Dimitar Dimitrov, Dimitar Tatarski, Dimo Dimov, Diyan Tonchev, Elena Rogleva, Hristiyan Hristov, Hristo Hristov, Iskra Nikolova, Ivelin Kirilov, Julieta Atanasova, Kalin Primov, Kristiyan Pamidov, Luboslav Lubenov, Nikolay Bankin, Nikolay Dimov, Pavlin Petkov, Petar Ivanov, Preslav Mihaylov, Rositsa Nenova, Rozalina Zaharieva, Ruslan Filipov, Stefany Boyanova, Stefany Ilieva, Stefka Vasileva, Svetlin Nakov, Teodor Kurtev, Todor Cholakov, Tonyo Zhelev, Tsvetan Iliev, Vasil Krumov, Vasko Viktorov, Ventsislav Petrov, Violeta Taseva, Yanitsa Vuleva, Yulian Linev, Zahariya Pehlivanova, Zhivko Nedyalkov
* ISBN: **978-619-00-1401-0** (9786190014010)
* Edition: **Faber Publishing, Sofia, Oct 2021**
* License: **CC-BY-SA**
* Source code: <https://github.com/SoftUni/Programming-Basics-Book-JS-EN>

# Preface

"**Programming Basics with JavaScript**" is the official book that introduces readers to writing **programming code** at a beginner's level (basic coding skills), working with the **development environment** (IDE), using **variables** and data, **operators** and **expressions**, working with the **console** (reading input data and printing output), usage of **conditional statements** (**if**, **if-else**, **switch-case**), **loops** (**for**, **while**, **do-while**) and working with **functions** (declaring and calling functions, passing parameters, and returning values). The book uses **JavaScript** programming language and **Visual Studio Code** development environment. The covered training material provides basic preparation for a deeper study of programming and prepares readers for the entrance exam in SoftUni.

|  |  |
| --- | --- |
| A yellow triangle with a exclamation mark  Description automatically generated | This book only gives you **the first steps to programming**. It covers very basic skills that you must develop for years, to reach a high enough level and start working as a programmer. |

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

## Who is This Book Intended for?

This book is suitable for **complete beginners in programming**, who want to try what programming is and learn the main constructions for writing programming code that is used in software development, regardless of programming languages and technologies used. The book provides a **solid basis** of practical skills that are used for further study in programming and software development.

## Why Did We Choose The JavaScript Language?

For this book, we chose **JavaScript**, because it is a **modern** language for high-level programming and at the same time it is easy to learn and suitable for **beginners**. As a use **JavaScript** is **widespread**, with a well-developed ecosystem, numerous libraries, and technological frameworks, and therefore gives many **prospects** for development. **JavaScript** combines the paradigms of procedural, functional, and object-oriented programming in a modern way with easy-to-use syntax. In the book, we'll use the **JavaScript language** and the **Visual Studio Code** development environment, which is available for free from Microsoft.

As we will explain later, **the programming language with which we start is not essential**, but we still need to use some programming language, and in this book, we have chosen **JavaScript**. You can find the principles and concepts explained here illustrated with other programming languages like C#, JavaScript, and Python [here](https://softuni.org/learn/resources/free-programming-books?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books).

## The Book in Other Programming Languages: Java, C#, C++, Python

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

* [Programming Basics with Java (English)](https://java-book.softuni.org/)
* [Programming Basics with Python (English)](https://python-book.softuni.org/)
* [Programming Basics with C# (English)](https://csharp-book.softuni.org/)
* [Programming Basics with JavaScript (English)](https://js-book.softuni.org/)
* [Programming Basics with JavaScript (Bulgarian)](https://js-book.softuni.bg/)
* [Programming Basics with Python (Bulgarian)](https://python-book.softuni.bg/)
* [Programming Basics with C# (Bulgarian)](https://csharp-book.softuni.bg/)
* [Programming Basics with Java (Bulgarian)](https://java-book.softuni.bg/)
* [Programming Basics with C++ (Bulgarian)](https://cpp-book.softuni.bg/)

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

## Programming is Learned by a Lot of Writing, Not Reading!

If anyone thinks they're going to read a book and learn to program without writing a code and solve tasks hard, they're delusional. Programming is learned with **practice**, with writing code every day and solving hundreds, even thousands of tasks, seriously and with perseverance, for years.

You need to **solve a lot of problems**, to make mistakes, to fix them, search for solutions and information on the Internet, to try, to experiment, to find better solutions, get used to the code, syntax, programming language, development environment, error search and debugging of non-working code, task reasoning, algorithmic thinking, breaking down problems in steps and implementing every step, gaining experience and lifting your skills every day, because learning to write code is just **the first step towards the profession of a software engineer**. You have a lot to learn!

We advise the reader, as a minimum, **to try all the examples from the book**, to play with them, to change and test them. Even more important than the examples **are the exercises** because they develop the practical skills of the developer.

**Solve all the tasks in the book**, because programming is learned by practice! The tasks after each topic are carefully selected to cover in-depth the covered learning material. The purpose of solving all tasks from all the topics covered is to give **complete writing skills to a program code** at the beginners level (as is the purpose of this book).

|  |  |
| --- | --- |
| A yellow triangle with a exclamation mark  Description automatically generated | **Solve all the exercises in the book**. Otherwise, you won't learn anything! Programming is learned by writing a lot of code and solving thousands of problems! |

# About Software University(SoftUni)

[The Software University (SoftUni)](https://softuni.org/learn/?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) is **the largest training center for software engineers in South-Eastern Europe**. Tens of thousands of students pass through the university every year. SoftUni was founded in 2014 as a continuation of the hard work of [**Dr. Svetlin Nakov**](https://nakov.com/) in training **skillful software engineering professionals** by a practical, contemporary, and high-quality education that combines fundamental knowledge with modern software technologies and a lot of practice.

The Software University provides **quality education**, **occupation**, **work**, and **the opportunity to acquire a diploma** for programmers, software engineers, and IT professionals. SoftUni builds an extremely successful lasting **relationship between education and industry**, collaborating with hundreds of software companies, **providing work and internships** to its students, providing quality specialists for the software industry, and directly meeting the needs of employers through the learning process.

## The SoftUni Interactive Classroom

SoftUni’s self-paced programs teach software developers worldwide in the innovative **Interactive Classroom** that combines **video lessons, coding sessions, examples, and exercises** in a single platform on the Web.

Using the **SoftUni Interactive Platform**, you **write, execute and test code directly into the browser** and your exercise solutions are automatically evaluated by the integrated **Judge system**. If you have difficulties with an assignment, you can **get assistance** in multiple ways: with automated hints and guidelines or through SoftUni’s **help center**. Try it now with a [**7-day free trial**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books).

This is how the **SoftUni Interactive Classroom** looks like:

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

## Video: SoftUni and SoftUni Judge

Watch a video lesson about SoftUni and SoftUni Judge here: <https://www.youtube.com/watch?v=IwLET8SDBE4>.

## The Automated Judge System

**The SoftUni Judge system** ([https://judge.softuni.org](https://judge.softuni.org/)) is an automated system on the Internet **that checks solutions to programming problems through a series of tests**. **Real-time** problem submission and verification: you send a solution and in seconds, get an evaluation. You earn points for each task depending on your answer. For a completely right one, you get the maximum points for the problem. With a partially correct solution, you receive a fraction of the points. And with a completely wrong answer, you get 0 points. This how the **SoftUni Judge** looks like:A screenshot of a computer program

Description automatically generated

**All problem in this book are available for testing in the SoftUni Judge system** and we strongly recommend that you test them once you have decided to know if you are missing something and whether your solution is working properly, as required by the problem.

Note also some **features of the SoftUni Judge system**:

* For each problem **the Judge system keeps the highest achieved result**. If you upload a solution with the wrong code or a weaker result than your previous one sent, the system won't take points away from you.
* The output results of your program are **compared** by the system strictly with the expected result. Any **redundant symbol missing a comma or space** may result in 0 points of the relevant test. **The output** that the Judge system expects is **described in the condition of each problem** and **nothing more should be added to it**.

**Example**: If a number is required to be printed in the output (e.g. **25**), do not display descriptive messages such as **The result is: 25**, but print exactly what is required, i.e. only the number.

The SoftUni Judge system is **available at any time** from its website: [https://judge.softuni.org](https://judge.softuni.org/).

* Use of the system is **free of charge**.

We are convinced that after a few problems are sent, you **will like to receive instant feedback** on whether the solution you wrote is true, and the judge system will become your favorite assistant when learning to program.

# How to Become a Software Developer?

Dear readers, many of you must have the ambition to become programmers, make a living with software development, or work in the IT sector. That is why we have prepared for you a **short guide "How to become a programmer"** to navigate you on the steps to this much-desired profession.

You can become a junior-level programmer after at least **1-2 years of daily learning and writing code**, solving thousands of programming problems, developing several practical projects resulting in a lot of experience in software development. It takes more than a month or two of effort! The profession of the software engineer requires a large amount of knowledge as a result of a lot of practice.

## Video: Become a Software Engineer – 4 Essential Skills

Watch a video lesson about SoftUni and SoftUni Judge here: <https://youtu.be/Ds5PD3UW57k>.

## The 4 Essential Skills of the Software Developers

There are **4 main skill groups** that all programmers must have. Most of these skills are preserved over time and are not significantly affected by the development of specific technologies (which change constantly). These are the skills that **every good programmer has** and which every rookie should aspire to obtain:

* code writing (20%)
* algorithmic thinking (30%)
* fundamental knowledge of the profession (25%)
* languages and development technologies (25%)

### Skill # 1 - Coding (20%)

Learning how **to write code** forms about 20% of the minimum skills required for a programmer to start work in a software company. The ability to code includes the following components:

* work with variables, conditionals, loops
* functions, methods, classes, and objects
* work with data: arrays, hash tables, strings

The ability to code **can be learned in a few months** of intensive studying and solving practical tasks by writing code every day. This book covers only the first part of the coding skill: **working with variables, conditionals, and loops**. The rest remains to be learned in subsequent trainings, courses, and books.

The book is only the beginning of a long-term study and a small step of your path to programming professionally. You won't be able to become a programmer without mastering the material from this book. You will lack programming fundamentals and it will become increasingly difficult to move forward. Therefore **pay enough attention to the basics of programming**: solve problems and write a lot of code for months until you learn **to easily solve all the problems in this book**. Then move on.

We pay special attention to the fact that **programming language doesn't have significant relevance** for one's coding skill. You can either code or not. If you can code in C#, then you'll easily switch to Java or C++, or any other language. These are the skills that each programming book for beginners starts with, including this one.

### Skill # 2 - Algorithmic Thinking (30%)

Algorithmic (logical, engineering, mathematical, abstract) thinking forms about 30% of the minimum skills for a start in the profession. **Algorithmic thinking** is the ability to break a task into a logical sequence of steps (algorithm) and to find a solution for each step, then to put them together in a working solution for the initial task. This is the most important skill that the programmer has.

How **to build algorithmic thinking**?

* Review **many (1000+) programming tasks**, the more diverse, the better. This is the recipe: reducing thousands of practical tasks, inventing an algorithm for them, and executing the algorithm, along with debugging errors along the way.
* Physics, mathematics, and/or similar sciences help, but they are not restraining! People with **engineering and technical inclinations** usually learn to think logically easily, because they already have the skills for solving problems, although not algorithmic.
* The ability **to solve programming tasks** (which requires algorithmic thinking) is extremely important for programmers. Many companies test for this skill specifically during job interviews.

This book develops **a beginner's level of algorithmic thinking**, but it is not enough to make you a good programmer. To become proficient in the professions, you will need to add **logical thinking skills and improve tasks** outside this book, such as **working with data structures** (arrays, lists, matrices, hash tables, tree structures) and basic **algorithms** (search, sorting, tree structures, recursion, etc.).

As you may guess, **the programing language does not matter** for the development of algorithmic thinking. To think logically is universal, even if it's not related only to programming. Because of their well-developed logical thinking, there is the misconception that all programmers are **smart people** and having a high IQ is a requirement for entering into the profession.

### Skill #3 – Fundamental Knowledge of The Profession (25%)

**Fundamental knowledge and skills** for programming, software development, software engineering and computer science form about 25% of the developer's minimum start-up skills. Here are the most important parts of these skills and knowledge:

* **basic mathematical concepts** related to programming: coordinate systems, vectors and matrices, discrete and indiscreet mathematical functions, end machines and state machines, concepts of combination and statistics, algorithm complexity, mathematical modeling, and others.
* **skills to program** - code writing, data work, use of conditional structures and loops, work with arrays, lists and associative arrays, strings, and word processing, working with streams and files, using program interfaces (APIs), working with a debugger, and others.
* **data structures and algorithms** - lists, trees, hash tables, columns, search, sorting, recursive, tree crawling, etc.
* **object-oriented programming** (OOP) – working with classes, objects, inheritance, polymorphism, abstraction, interfaces, data encapsulation, exception management, design templates.
* **functional programming** (FP) - working with lambda functions, high order functions, functions that return a function as a result, closing a state in a function (closure), and more.
* **databases** - relational and non-relational databases, database modeling (tables and links between them), SQL query language, object-relational data access (ORM) technologies, transactionality, and transaction management.
* **network programming** - network protocols, network communication, TCP/IP, concepts, tools, and technologies from computer networks.
* **client-server** interaction, communication between systems, back-end technologies, front-end technologies, MVC architectures.
* **back-end development technologies** - web server architecture, HTTP protocol, MVC architecture, REST architecture, web development frameworks, templating engines.
* **web front-end technologies (client development)** - HTML, CSS, JS, HTTP, DOM, AJAX, back-end communication, REST API call, front-end frameworks, basic design and UX (user experience) concepts.
* **mobile technologies** - mobile applications, Android and iOS development, mobile user interface (UI), server logic call.
* **built-in systems** - microcontrollers, digital and analog input and output control, sensor access, peripheral control.
* **operating systems** - work with operating systems (Linux, Windows, etc.), installation, configuration and basic system administration, process handling, memory, file system, users, multitasking, virtualization, and containers.
* **parallel programming and asynchronousness** - thread management, asynchronous tasks, promises, common resources, and access synchronization.
* **software engineering** - source control systems, development management, task planning and management, software development methodologies, software requirements and prototypes, software design, software architectures, software documentation.
* **software testing** - unit testing, test-driven development, QA engineering, error reporting and error trackers, test automation, building processes, and continuous integration.

We must emphasize once again that the choice of **the programming language is not a significant factor** in acquiring these skills.. They accumulate slowly, over many years of practice in the profession. Some knowledge is fundamental and can be learned theoretically, but it takes years of practice to fully understand them in depth.

Fundamental knowledge and skills for programming, software development, software engineering, and computer science are taught during the [**Software Developer Program**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) and [**open courses**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books). Working with a variety of software libraries, APIs, frameworks, and software technologies and their interaction gradually builds this knowledge and skills, so do not expect that you will understand them from a single course, book or project.

**Basic knowledge in the areas listed above** is usually sufficient to start working as a programmer. You will gain a deeper understanding of the concepts from the technologies and development tools you use in your day-to-day work.

### Skill #4 - Programming Languages and Software Technologies (25%)

**Programming languages and software development technologies** form about 25% of the developer's minimum skills. They are the most voluminous to learn, but they change most quickly over time. If we look at **job advertisements** from the software industry, there are often all sorts of words mentioned (such as those listed below), but in fact, the ads silently **imply the first three skills**: to code, to think algorithmically, and to be proficient at the foundation of computer science and software engineering.

**The choice of a programming language is essential** for acquiring technological skills.

* **Note**: only for this 25% of the profession does programming language matter!
* **For the remaining 75% of skills, language does not matter** and these skills are time-resilient and transferable between different languages and technologies.

Here are some commonly used software development stacks sought by software companies (up-to-date as of January 2018):

* **JavaScript** (JS) + CMO + PH + databases + MongoDB or MySQL + HTTP + web programming + HTML + CSS + DOM + jQuery + Node.js + Express + Angular or React
* **C#** + CMO + P + classes from .NET + DATABASE SQL Server + Entity Framework (EF) + ASP.NET MVC + HTTP + HTML + CSS + JS + DOM + jQuéry
* **Java** + Java API classes + CMO + AP + databases + MySQL + HTTP + web programming + HTML + CSS + JS + DOM + jQuery + JSP/Servlets + Spring MVC or Java EE / JSF
* **PHP** + CMO + databases + MySQL + HTTP + web programming + HTML + CSS + JS + DOM + jQuery + Laravel / Symfony / other MVC framework for PHP
* **Python** + CMO + PH + databases + MongoDB or MySQL + HTTP + web programming + HTML + CSS + JS + DOM + jQuery + Django
* **C++** + CMO + STL + Boost + native development + databases + HTTP + other languages
* **Swift** + MacOS + iOS + Cocoa + Cocoa Touch + XCode + HTTP + REST + other languages

If the words listed above seem scary and incomprehensible to you, then you are quite at the beginning of your career and have **years to learn** until you reach the profession of a **software engineer**. Don't worry, every programmer goes through one or more technological stacks and has to study a **set of interconnected technologies**, but the most important skill is **writing programming logic (to code)** and **thinking algorithmically** (to solve programming problems). Becoming a good software engineer is impossible without those skills.

### Programming Language Doesn't Matter!

As we already made clear, the technical skills, determined by mastering a specific programming language and technology, amount to about **10-20% of a software developer’s overall skillset**.

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

Let us state once again that **the choice of programming language (mostly) does not matter** - you just need to learn to program. This process starts with **coding** (by reading this book or enrolling in a [Software Developer program](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books)), continues with mastering more **complex programming concepts** (like data structures, algorithms, OOP, and FP), and includes using **fundamental knowledge and skills for software development, software engineering, and computer science**.

You will need to know a **specific programming language**, program libraries (APIs), frameworks, and software technologies (front-end UI technologies, back-end technologies, ORM technologies, etc.) once you start working on a software project.

This book uses the **Python** language, but it is not essential and can be replaced by Java, C#, JavaScript, PHP, C++, Ruby, Swift, Go, Kotlin, or any other language. To master **the profession of "software developer"** it is necessary to learn **to code** (20%), learn to **think algorithmically** and **solve problems** (30%), have a **fundamental knowledge of programming and computer science** (25%) and know a **specific programming language and the technologies relevant to it** (25%). Be patient, for a year or two, all this can be achieved at a good starting level, as long as you are serious and diligent.

# The History Behind This Book

The main engine and project manager for the creation of the current **free programming book for beginners** with open source is [**Svetlin Nakov, PhD**](https://nakov.com/).

In the beginning, these chapters had a wider reach and included more theory, but in 2016 Svetlin Nakov, PhD completely revised them, renovated, simplified, and **directed them to be more practical**. This is how the **learning content core of this book was created**.

On the principle of free software and free knowledge, Svetlin Nakov led a **team of volunteers** and started this open-source project, initially to create a book based on programming with the C# language and later with other programming languages.

## Authors Team

This book, "**JavaScript Basics**", was developed by a broad author's team of **volunteers** who took their time to give you this systematic knowledge and guidance at the start of programming. List of all book authors and editors (alphabetically):

**Boncho Vulkov, Christian Hristov, Dimitar Dalev, Elena Rogleva, Hristo Minkov, Ivelin Arnaudov, Julieta Atanasova, Kristian Marianov, Martin Chaov, Nikolay Bankin, Nikolay Kostov, Pavel Kolev, Petar Ivanov, Stiliyan Kangalov, Svetlin Nakov, Ventsislav Petrov, Zahariya Pehlyavanova, Zdravko Kostadinov**

The book is based on its initial C# variant ([Programming Basics with C#](https://csharp-book.softuni.org/)), which is developed by a large team of authors that has a significant contribution to the current book. Below is a list of the additional book contributors (editors, translators, others):

Aleksander Krastev, Aleksander Lazarov, Aleksander Peev, Alen Paunov, Angel Dimitriev, Ariet Motzeva, Daniel Tsvetkov, Denis Milanov, Dimitar Dimitrov, Dimitar Tatarski, Dimo Dimov, Diyan Tonchev, Elena Rogleva, Hristiyan Hristov, Hristo Hristov, Iskra Nikolova, Ivelin Kirilov, Julieta Atanasova, Kalin Primov, Kristiyan Pamidov, Luboslav Lubenov, Nikolay Bankin, Nikolay Dimov, Pavlin Petkov, Petar Ivanov, Rositsa Nenova, Rozalina Zaharieva, Ruslan Filipov, Stefany Boyanova, Stefany Ilieva, Stefka Vasileva, Svetlin Nakov, Teodor Kurtev, Todor Cholakov, Tonyo Zhelev, Tsvetan Iliev, Vasil Krumov, Vasko Viktorov, Ventsislav Petrov, Violeta Taseva, Yanitsa Vuleva, Yulian Linev, Zahariya Pehlivanova, Zhivko Nedyalkov.

Cover design: Marina Schiderova.

## Official Website of The Book

The current book on **The basics of JavaScript programming for beginners** is available for free use on the Internet from:

[**https://js-book.softuni.org**](https://js-book.softuni.org/)

This is **the official website of the book** and there will be uploaded its latest version. The book has been similarly translated to other programming languages listed on its website.

## Forum for Your Questions

Ask your **questions to this book** on Programming Basics in the **SoftUni's Reddit Community**:

[**https://www.reddit.com/r/softuni/**](https://www.reddit.com/r/softuni/)

In this discussion subreddit, you will receive a **free adequate answer on any questions from the curriculum content of this textbook**, as well as on other programming issues. SoftUni's teachers, assistants, and mentors also answer your questions constantly.

If you have a specific question, such as why a program doesn't run, that you've been stuck with for a few hours, **ask it in the subreddit** and you'll get an answer. You will be surprised how well-intentioned and responsive the users of the SoftUni’s subreddit are.

## License and Distribution

The book is distributed **for free** in electronic format under an open CC-BY-NC-SA license [**CC-BY-NC-SA**](https://creativecommons.org/licenses/by-nc-sa/4.0/).

**The source code** of the book can be found on GitHub: [**https://github.com/SoftUni/Programming-Basics-Book-JS-EN**](https://github.com/SoftUni/Programming-Basics-Book-JS-EN).

International Standard Book Number (ISBN): **978-619-00-1401-0**.

## The Book That Helps Teachers

If you are a **teacher of programming**, informatics, or information technology or want **to teach programming**, this book gives you more than a well-structured learning material with many examples and tasks. **Free of charge** with the book you receive **quality educational content** for teaching in school, in the **Bulgarian language**, by the school requirements:

* **Educational presentations** (PowerPoint slides) for each learning topic tailored to 45-minute hours in schools – free of charge.
* Well-designed **tasks** for class and homework, with detailed conditions and sample entrance and exit – free of charge.
* **An automated task and homework verification system** (Online Judge System) to be used by students, also free of charge.
* **Video lessons** with methodological instructions from the free course for programming teachers, which is held regularly by the SoftUni Foundation.

## Bug Reports

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

[**https://github.com/SoftUni/Programming-Basics-Book-JS-EN/issues**](https://github.com/SoftUni/Programming-Basics-Book-JS-EN/issues)

We do not promise that we will correct everything you send us, but we are willing to constantly improve the quality of this book so that the reported unmistakable errors and all reasonable suggestions will be addressed.

## Enjoy Your Reading!

And be sure **to write code** in large quantities, **try the examples** from each topic and, above all, **solve the tasks from the exercises**. With just reading, you won't learn to program, so make sure you put time into solving exercises!

# Chapter 1. First Steps in Programming

In this chapter, we are going to find out **what programming** is at its core. We will get familiar with the idea of programming languages and explore the **environments for software development** (Integrated Development Environment – IDEs); we'll see how to work with them, in particular with **Visual Studio Code**. We will write and execute our **first program** written in the programming language **JavaScript** and then we'll exercise on a few tasks: we will create a console-based program and a Web application. We will learn how to use **SoftUni's Judge System** to check the correctness of our solutions to the problems in this book. Finally, we'll look at some typical mistakes made during code writing and we'll learn how to avoid them.

## What Does "To Program" Mean?

To program means to give instructions to the computer, for example, play a certain sound, print something on the screen, or multiply two numbers. When there's a series of commands, we have a computer program (script). The computer program's text is called **program code** (**source code** or just **code**).

## Computer Programs

**Computer programs** represent a **series of commands** written in a particular **programming language**, like Python, C#, Java, JavaScript, PHP, C, C++, Go, etc. To write commands, we need to be familiar with the **syntax and semantics of the language** we'll be using, in our case – **JavaScript**. In this book, we'll cover programming in general, as well as JavaScript's particular syntax and semantics. We'll examine each step of writing code, starting with the simplest and eventually reaching more complex programming constructs.

### Algorithms

Computer programs usually execute some sort of algorithm. **Algorithms** are a series of steps used to complete a task and achieve an expected result - something like a recipe. For example, if we need to fry some eggs, we follow a certain recipe (algorithm): we heat some oil in a pan, then we break the eggs, we wait until they're cooked, and finally, move the pan away from the heat. Similarly, in programming, **computer programs execute algorithms** – a series of commands needed to complete a certain task. If we want to print a series of numbers in ascending order, for example, we need an algorithm. It will go through all the numbers, find the smallest one and print it, then go through the rest of them and do the same until there are no more numbers left. To make creating programs, writing program code (commands), executing it and other operations that have to do with programming more convenient, we need a **development environment** (IDE) like Visual Studio Code.

### Programming Languages, Compilers, Interpreters and Development Environments

**Programming languages** are artificial languages (syntax for expression), designed to **issue commands** which we expect the computer to read, process, and execute. With the help of programming languages, we write a series of commands (**programs**) that **tell the computer what to do**. Execution of computer programs can be achieved by using either a **compiler** or an **interpreter**.

**The compiler** translates code from a programming language into **machine code** and for each construct (command) in the code, it chooses an appropriate, predefined fragment of machine code while simultaneously checking the program's text for **errors**. Together, all the compiled fragments represent the original program translated in machine code, exactly how the computer's microprocessor is expecting it. After it's been compiled, the program can be directly executed by the microprocessor in cooperation with the operating system. Compiled programming languages **compile the program** before executing it and find syntactic errors (incorrect commands) during compile time. Languages like C++, C#, Java, Swift, and Go all work with compilers.

Some programming languages don't use a compiler but are **directly interpreted** by specialized software called an interpreter. **The interpreter** is a **program that executes programs**, written in some programming language. It executes the program's commands in sequence and understands not only single commands and series of commands but also other language constructs (checks, iterations, functions, etc.). PHP, Python, and **JavaScript** are languages that work with an interpreter and are executed without the need to be compiled. Due to lack of preliminary compilation, interpreted languages' errors are found during run time, after the program has already started working.

Overall, **compiled languages work more slowly than their interpreted counterparts**. The difference in execution speed is sometimes insignificant, but in other cases massive. In interpreted programming languages **we can change code more quickly and re-execute** (to remove a mistake, for example), while compiled languages are slower after a code change has been introduced, due to the need for compilation.

**The development environment** (Integrated Development Environment – **IDE**) combines traditional instruments for software development. In the development environment we write code, compile and execute programs. Development environments incorporate a text editor for writing code, a **programming language**, **a compiler or interpreter**, an **execution environment** where our program is executed, a **debugger** that traces our program and looks for errors, **instruments for design, and user interface** as well as other instruments and add-ons.

**Development environments** are convenient because they incorporate everything necessary to develop a program, so there's no need to leave the environment. If we don't use a development environment, we'll have to write code in a text editor, compile it with a command from the console, execute it with another command from the console and write additional commands, when needed, which will cost a lot of time. This is why most programmers use IDEs for everyday work.

Programming with **JavaScript** is usually done in the development environment **Visual Studio Code** which is developed and freely distributed by Microsoft. You can download it from: <https://www.visualstudio.com/downloads/>. Alternatives to Visual Studio Code are **WebStorm** (<https://www.jetbrains.com/webstorm/>), **Atom** (<https://atom.io/>), and others. In this book, we'll be using the development environment **Visual Studio Code**.

### Low-Level, High-Level Languages and Runtime Environments

A program is, in essence, a **set of instructions** that ask the computer to carry out certain tasks. They are entered by the programmer and **executed unconditionally by the machine**.

There are different types of **programming languages**. **Lowest-level** languages can be used to write the very **instructions commanding the processor** - **Assembler** is one such language. Higher-level languages can be utilized to create an operating system, drivers for managing hardware (video card drivers, for example), web browsers, compilers, engines for game graphics (game engines), and other system components and programs. Even higher-level languages like **JavaScript**, **C#**, and **Python** are used to create application software, like programs for reading mail or chatting.

**Low-level languages** command hardware directly and require a lot of effort and a vast number of commands, to carry out a certain task. **Higher-level languages** require less code to achieve the same result but have no direct access to hardware. They are used to develop application software like web and mobile applications.

Most of the software we use daily like music players, video players, GPS programs, etc. is written on high-level **languages for application programming** like Python, JavaScript, C#, Java, C++, PHP, etc.

**JavaScript is an interpreted language** which means we write commands and they're executed immediately after running the program. As a result of that, we will realize we've made a mistake in our code only after we've started our program and reached the incorrect command. In cases like these, **IDEs** like Visual Studio Code are extremely helpful as they check our code while we're writing it and alarm us of potential issues. After we've written our code and want to test it, we can save it in a file with extension **.js**.

Most **programming languages** are specialized in the development of certain types of applications – Desktop for Windows and Mac, mobile apps, server applications, applications that command smart gadgets, and many more. **JavaScript** is one of the few languages which allow you to create an application in every field you can think of – from websites and mobile apps to desktop applications.

The most popular JavaScript **interpreters** are the web browsers you use every day – Chrome, Firefox, Internet Explorer, etc. When loading your favorite website, it will likely contain **JavaScript** files which will execute upon opening the site and make your viewing experience of the site much more pleasant and dynamic. It's also very often the case that if the site has dropdown menus, animations, user registration, and annoying adverts, they are realized exactly by using the **JavaScript** language.

You'll often hear that a certain piece of code is run on the **client-side**, which means that the **JavaScript** code is executed in your **browser**, which plays the role of client or receiver. But a receiver would have no purpose to exist without a transmitter. In the world of technology, these transmitters are called servers. You can imagine servers as extremely powerful computers, which many people have access to. Every site uses a similar server as its base – i.e. files that enable the site to look the way you see it – **pictures**, **texts**, **JavaScript files**, are located on a certain server. Your **browser** (client) connects to the **server** where the site you're visiting is located, and the server, in turn, **sends** back the necessary files for your favorite website to visualize in front of your eyes. Your browser is like a car radio where you listen to your favorite program, and the server is the building that transmits that program, although the communication follows a slightly different route in this case.

The other popular **JavaScript interpreter** is **NodeJS**. You can imagine it's like an application you install on your computer which enables it to understand JavaScript the same way your browser does. That way you can execute **JavaScript** code directly on your computer, without the need for a browser. As we just mentioned, servers are just more powerful computers. They start understanding **JavaScript** in the same way your computer does – by having **NodeJS** installed. You can install **NodeJS** from their official website [https://nodejs.org](https://nodejs.org/) completely free of charge, by simply following the instructions.

## Computer Programs - Execution

As we already mentioned, a program is a **series of commands**, in other words, it describes a series of calculations, checks, iterations, and numerous other operations, which aim at producing a result. The program is written in text format and the text is called **source code**. It's saved in a file with extension **.js** (**main.js**, for instance), and then it can be executed on your **browser** or the **console** with the help of **NodeJS**. In a moment we'll have a look at both options.

### Computer Programs – Problems

Let's start with a very basic example of a short **JavaScript program** that we'll run directly in your web browser (it supports JS without the need to install additional software).

#### Problem: a Program, Which Notifies the User

Our first program will consist of a single **JavaScript command**, which notifies the user they've won 1 000 000 dollars, as is often the case when you're browsing through a website with a lot of spam and adverts:

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

We can run our program by using our browser's **JavaScript console**. In **Chrome**, for example, we can pull that up by pressing **[F12]** and we can then write our code in the **[Console]** window.

A screenshot of a computer

Description automatically generated

The result is something like this: a **modal popup message** in your browser:

A screenshot of a computer

Description automatically generated

#### Problem: a Program, Which Spams the User with Notifications

We can increase the previous program's complexity by creating numerous command which repeat in sequence and notify the user they've won the lottery:

for (i = 0; i < 10; i += 1) {

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

}

In the example above we make the computer display notifications one after another until they become 10. The result is one very irritated user: A screenshot of a computer

Description automatically generated We will learn how iterations (loops) in programming work in the chapter "[Loops](https://js-book.softuni.org/chapter-05-loops.html)", but for now, let's assume that they just repeat a command many times.

#### Problem: Program, Which Converts BGN (Bulgarian Levs) into Euro

Let's have a look at another simple program that asks the user for an amount of money in levs (Bulgaria's currency)(whole number), makes sure that what's been entered is a number, converts it into euro (by dividing it by the euro exchange rate) and prints the result. This is a program of 3 consecutive commands.

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

let leva = parseInt(myMoney);

let euro = leva / 1.95583;

console.log(euro);

If we run this program in the browser's JavaScript console, we'll receive something like this:

A screenshot of a computer

Description automatically generated

We looked at **three examples of computer programs**: single command, series of commands in a loop, and a sequence of four commands. Let's now move on to the more exciting bit: writing our programs in **JavaScript** and executing them outside the browser.

## How do We Write a Console Program?

Let's go through the **steps for creating and executing a computer program** that uses a text console (window for entering and receiving text) to read and write its data. Such programs are known as **console-based**. Before we do that though, we need to **install and set up our development environment**, where we'll write and execute the **JavaScript** programs from this book and the exercises that go along with it.

## Development Environment (IDE)

As previously mentioned, we need a **development environment (IDE)** to program. The IDE is a program editor, where we can write, compile and execute our program code, discover our mistakes, correct them and start the program again.

* For programming in JavaScript, we use the Visual Studio Code environment, suitable Windows, Linux, and macOS operating systems.
* If we are programming in **Java** we can use **IntelliJ**, **Eclipse**, or **NetBeans**.
* If we are programming in **Python** we can use **PyCharm**.

### Installing Visual Studio Code

In the following paragraphs, we go through the steps for installing **Visual Studio Code** (version 1.19.1) in great detail. After we download and start the installation file, the following screen should appear:

A screenshot of a computer program

Description automatically generated

Press the **[Next]** button and then agree to the terms and conditions:

A screenshot of a computer program

Description automatically generated

Visual Studio Code starts installing and once it's finished, we're almost ready to start writing code. After you start VS Code, a window similar to the one below should appear:

A screenshot of a computer

Description automatically generated

Now is the time to make our **development environment** as pleasant as possible. **Visual Studio Code** is famous for being the IDE with the most options for personalization. Usually, this personalization comes in the form of **add-ons** (extensions). A list of all add-ons can be found on the official website: <https://marketplace.visualstudio.com/>. Just keep in mind that most of these extensions are specific to a particular programming language.

For a start, we recommend installing a couple of extensions that will drastically improve your JavaScript code writing:

* **Beautify** - <https://marketplace.visualstudio.com/items?itemName=HookyQR.beautify> – this is an extension that helps you order your code and keep it looking clean.
* **JSHint** - <https://marketplace.visualstudio.com/items?itemName=dbaeumer.jshint> – as we mentioned, JavaScript is an interpreted language and core errors would appear only after running the program. This extension scans and alarms for potential issues while the code is written and before running the program.

They can be installed either via the given links or by going through the following steps in **Visual Studio Code**:

1. In the leftmost panel, open the lowest tab – Extensions.
2. In the search field enter the name of the extension you'd like to install.
3. Choose the most appropriate option from the results.
4. Then press the **[Install]** button.
5. And restart **Visual Studio Code**.

A screenshot of a computer

Description automatically generated

That's it. We're ready to start working with **Visual Studio Code** and **JavaScript**.

### Online Development Environments

There are also **alternative environments to develop online**, directly in your web browser. They aren't very easy to use but if you have no other option, you can start your education with them and install **Visual Studio Code** later. Such a site for online JavaScript development is **JSBin** - <https://jsbin.com/?js,console>.

A screenshot of a computer

Description automatically generated

Alternatively, we can do quick tests directly in our browser by pressing **F12** but in general, this is an option that works for very few lines of code. Here's an example:

A screenshot of a computer

Description automatically generated

## Problem: Creating a Console Application 'Hello JavaScript'

Let's go back to our console program. We already have Visual Studio Code and we can start it. Then create a new **JavaScript file**: [**File**] → [**New File**]:

A screenshot of a computer program

Description automatically generated

It's crucial to save our file from [**File**] → [**Save**], and don't forget the **.js** extension! We also need to give it a meaningful name like **helloJS**:

A screenshot of a computer

Description automatically generated

### Writing Program Code

Writing code in **JavaScript** requires no additional preparation – creating a file with the **.js** extension is all we need. So let's directly proceed with writing our first line of code. We'll write the following command:

console.log("Hello JavaScript!");

Here's what our program should look like in **Visual Studio Code**:

A screenshot of a computer

Description automatically generated

The console.log('Hello JavaScript!') command means printing (log(…)) on the console (console) the message Hello JavaScript! which we need to surround with quotes, to signify it is text. At the end of every **JavaScript** command, we place a semicolon ; which indicates the end of the command (meaning it doesn't continue on the next row). Although the latter is not compulsory, it's considered good practice which helps find problems more easily.

Important: the alert(…) and prompt(…) commands we used, work only in the web browser and are not available in VS Code console applications. If you try using them, you will receive an error.

This command is very typical for programming: we signify we need a certain **object** (the console, in this case) to be found and a **certain action** ( printing something, passed within brackets, in this case) to be performed on it. In more technical terms, we call the method log(…) from the class console and pass as a parameter to it the text literal Hello JavaScript!.

### Starting The Program

To start the program we need to hit **[F5]**. The result will be displayed on the console, which, for our convenience, will open directly at the bottom of **Visual Studio Code**:

A screenshot of a computer

Description automatically generated

Notice we run the program with **[F5]**, not **[Ctrl + F5]**. If we use the latter, we won't be able to see the result because the program will execute very quickly and then disappear.

The output from the program is the following text message:

Hello JavaScript!

The "**Debugging with inspector protocol . . .**" and **Debugger listening on...** messages are displayed as additional information at the very top of Visual Studio Code's console after the program starts executing. This gives us additional information about the execution which, for the time being, we will ignore.

### Testing in The Judge System

Testing the solutions to the problems in this book is completely automated via the **Judge System's website**: <https://judge.softuni.org/>. The solutions are judged in real-time by the system. Each solution goes through a series of tests that are hidden; every test that passes grants the user certain points.

The program we just wrote can be tested here: <https://judge.softuni.org/Contests/Practice/Index/926#0>.

Each piece of **JavaScript** we'd like to test in the **Judge System** needs to be surrounded by the following lines of code:

function solve() {

// we place our code here

}

In other words, if we'd like to test the program we just wrote in the system, it will look like this:

function solve() {

console.log("Hello JavaScript");

}

We need to place the entire source code of the program in the black field and choose **JavaScript code**, as shown in the screenshot below:

A screen shot of a computer code

Description automatically generated

Then we can send our solution by clicking the **[Submit]** button. The system returns a result in a few seconds which is displayed in the table of submitted solutions. If necessary, we can hit the **[Refresh]** button at the upper right-hand corner of the table of submitted solutions:

A screenshot of a computer

Description automatically generated

In that table, the Judge System will display one of the following **possible results**:

* A number of points (between 0 and 100), when the submitted code is compiled successfully (there are no syntactic errors) and can be tested.
  + If our solution is **completely correct**, all tests are marked in green and we receive **100 points**.
  + If our solution is **incorrect**, some tests are marked in red and we receive **less than 100** or **0 points**.
* If there are syntactic errors in our program, we receive a **compile-time error message**.

## Executing Code in The Browser Using HTML + JS

So far we've seen how to make and execute a console program. Let's now look at how we can write code, which runs in our browser. All the websites you visit are created in a very similar fashion.

The principle is very similar to what we just did. The only difference is that when creating a new file, we use the **.html** extension instead of **.js**. All that's left is to enclose our code with a opening <script> and closing </script> **HTML** tag. We enclosed our code in much the same way when submitting our code in the Judge System. Here's what our code in **Visual Studio Code** should now look like:

A screenshot of a computer

Description automatically generated

With this approach, now we just need to find the **helloJS.html** file in the location we've saved it, and double click it. It will load in the browser but to see the result, we need to hit **F12**, which will open the browser's console. Our command prints on the console, so it makes sense that we need to bring it up to see the result.

A screenshot of a computer

Description automatically generated

Now that you've **learned how to execute programs**, you can test the sample notification programs you have above. Have fun with them, try out different things. Try changing them and playing with them. Swap the console.log("Hello JavaScript"); **command for console.error("Error occurred");** and start your program. Note that notification programs can only be executed in our browser and when we try running them from the console, we receive an error. This is because the console doesn't support notification via visual element, like alert.

## Typical Mistakes in JavaScript Programs

One of the usual mistakes beginners make is mixing up **capital and lowercase letters**. However, they matter when we call commands and can impede proper functioning. Here's an example of such a mistake:

function solve() {

Console.Log("Hello JavaScript");

}

In the example above, Console is written incorrectly and the capital letter needs to be changed to lowercase. Is there another similar mistake in the same example?

A missing **quotation mark** or **closing bracket** can also cause issues – the program will either **function improperly** or not execute at all. Such mistakes are hard to notice, especially with larger amounts of code. Here's an example:

function solve() {

console.log("Hello JavaScript);

}

This program will return an **error after it's started executing** and even before that, the code will be underlined by the extensions which monitor syntax, to draw the programmer's attention to the missed closing bracket.

A screenshot of a computer

Description automatically generated

## What Have We Learned from This Chapter?

First of all, we learned **what programming is – issuing commands written in a programming language** that the machine can understand and carry out. We also found out what a **computer program** is – a **series of commands** that aim to achieve a certain result. We gained some basic knowledge of the **JavaScript programming language** and learned how to create **simple console and web programs using Visual Studio Code**. Then we examined JavaScript's program code structure. We looked at printing on the console with the function console.log() and starting the program with **[F5]**. And last but not least, we now know how to test our code in **SoftUni's Judge System**.

Top work! Let's now tackle the **exercises**. You do remember that learning how to program involves a lot of code writing and problem-solving, right? Let's do just that and put what we've learned into practice.

## First Steps in Programming – Problems

Welcome to the exercises. We are now going to write a few console applications, which will help us make a few more steps into programming. Then we will show you how to program something more complex – programs with graphical and web user interfaces.

### Problem: Expression

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

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

Note: **you are not allowed to previously calculate the value** (for example with Windows Calculator).

### Hints and Guidelines

Create a new **JavaScript file** and name it expression.js. Then we need to write the **code** which will calculate the numeric expression above and print its value on the console. Pass the expression to the console.log(...) command by writing it within its brackets:

A close-up of numbers

Description automatically generated

Start the program with **[F5]** and check whether the result matches the one from the picture:

A screenshot of a computer

Description automatically generated

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/926#1>.

**Note**: Don't forget to enclose your code in the solve() function:

function solve() {

// your code

}

A screenshot of a computer

Description automatically generated

### Problem: Nums 1...20

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

### Hints and Guidelines

Create a new **JavaScript file** and name it nums1to20.js. Inside the file write 20 console.log() commands, each on a separate line, to print the numbers from 1 to 20 one after another. Some of you may be wondering if there is a cleverer way. Don't worry, there is, but we will mention it later on.

A screenshot of a computer

Description automatically generated

Let's now **start the program** and check whether the result is what we're expecting:

1

2

…

20

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/926#2>. Did you receive 100 points? If not, think of what you may be missing. Then try and think of a **cleverer** way to write this program, to avoid repeating the same command so many times. Look up information for "[**for loop JavaScript**](https://www.google.com/search?q=for+loop+JavaScript)" on the Internet.

### Problem: Triangle of 55 Stars

Write a **JavaScript** console program that **prints a triangle of 55 stars** on 10 lines:

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

### Hints and Guidelines

Create a new **JavaScript file** and name it triangleOf55Stars.js. Inside we need to write code that prints our triangle, for example by using 10 print commands, like in the example below:

console.log("\*");

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

…

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/926#3>. Try and **improve your solution** to avoid repeating the same command so many times. Can this be achieved with a for-loop? Did you manage to invent a better solution (with a for-loop, for example) to the previous problem? The current problem can be solved with a similar, but a little more complex approach (a loop within another loop). It's completely fine if you can't figure it out, you'll remember this problem when we learn about loops in a few chapters.

### Problem: Rectangle Area

Write a **JavaScript program** that **receives two numbers a and b, then calculates and prints** the area of a rectangle with sides **a** and **b**.

### Sample Input and Output

| **a** | **b** | **area** |
| --- | --- | --- |
| 2 | 7 | 14 |
| 7 | 8 | 56 |
| 12 | 5 | 60 |

### Hints and Guidelines

Create a new **JavaScript file**. For now, programs of similar type will be tested only in the **Judge System**, which has a built-in mechanism for passing input data to the program. To **receive the two numbers**, we need to declare our request by changing the enclosing code (the solve() function) we got used to writing:

A screenshot of a computer

Description automatically generated

Did you notice the change? We placed additional square brackets within the brackets of the function, inside which we described what data we're expecting to receive – in this case the numbers a and b, set as an array. What's left is to finish the program so it can calculate the rectangle's area and print it. Pass the product of a and b to the familiar console.log() command. In programming, we multiply using the \* operator.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/926#4>.

### \* Problem: Square of Stars

Write a **JavaScript** console program that **receives a whole positive number N** and **prints** a square of N stars on the console, like in the examples below.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 3 | \*\*\* \* \* \*\*\* | 4 | \*\*\*\* \* \* \* \* \*\*\*\* | 5 | \*\*\*\*\* \* \* \* \* \* \* \*\*\*\*\* |

### Hints and Guidelines

Create a new **JavaScript console program**:

A screenshot of a computer

Description automatically generated

Finish the program so that it prints a square made up of stars. You may need to use for-loops. Search for additional information on the Internet.

**Caution**: this task is more difficult than the rest and is presented to you at this point purposefully. It's marked with a star, To provoke you to **look for information on the Internet**. This is one of the most important skills you have to develop while you're learning to program. This is what you'll be doing every day if you work as a developer, so don't be scared, try it out. If you're having difficulties, you can also ask for help in the SoftUni Reddit: <https://www.reddit.com/r/softuni/>.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/926#5>.

## Console, Graphical and Web Applications

With **console applications**, as you've already noticed, all operations for reading input and printing output are done on the console. The input data is entered on the **console** and then it is read by the application; the console is also used to print the output data after or during the runtime of the program.

While a console application **uses the text console**, web applications use a **web-based user interface**. To execute them, two things are needed – a web server and a web browser. The browser plays the main role in the visualization of data and interaction with the user. Web applications are much more pleasant for the user, they look better, and a mouse and touch screen can be used (for tablets and smartphones). Programming stands under all of that, of course. This is why we need to learn how to program and we have already made our first tiny steps toward achieving that.

Graphical (GUI) applications have a **visual user interface**, directly on your computer or mobile device, without a web browser. Graphical applications (also known as desktop applications) **contain one or more graphical windows** where certain **controllers** are located (text fields, buttons, pictures, tables, and others), which **allow dialog** with the user to be more intuitive. Similar to them are mobile applications on your smartphone or tablet: we use forms, text fields, buttons, and other controls, and we manage them with program code. This is why we learn to write code now: **code is everywhere in software development**.

## Exercises: Web Applications

We will now build a basic **web application** and after that, a simple graphical application, To have a look at what we will be able to create once we progress with programming and software development. We are not going to examine the techniques and constructions used in much detail. Rather, we are just going to have a peek at the arrangement and functionality of what we've created. After we progress with our knowledge, we will be able to develop bigger and more complex software applications and systems. We hope that the examples given below will **boost your interest**, rather than make you give up.

## Problem: Web Application Numbers Summator'

Write a **Web application** that **calculates the sum of two numbers**.

After the numbers are entered in the first text fields, and the button **[Calculate]** is clicked, calculate their sum and display the result in the third text field. For our application, we'll use **HTML technology** which in combination with the **JavaScript programming language**, allows for the creation of **web applications and sites**, in the **Visual Studio Code** programming environment.

**HTML** is a descriptive language that is used to declare the content of a website. Without going into too much detail, we'll just mention that **HTML** is based on the use of combinations of **tags** to visualize and give **semanticity** to the content. In one of our previous examples, we already created an **HTML page** by utilizing the <script> tag.

Notice that we'll be creating a **web-based application**. This is an application accessible through a web browser, exactly like your favorite webmail or news site. The web application will have a server-side (Back-end), written in **JavaScript**, and a client-side (front-end), written in **HTML**. The Web application is expected to look similar to the following:

A screenshot of a computer

Description automatically generated

As opposed to console applications that read and write data in the form of text on the console, Web applications have a **Web-based user interface**. Web applications are **loaded from some Internet address** (URL) through a standard web browser. Users write input data on a page, visualized by the web browser, the data is processed on a web server and the results are shown again on a page in the web browser. As we mentioned we'll be using **HTML and JavaScript** for our web application. Other technologies which support creating **web applications** are **ASP.NET MVC technology**, **PHP technology**, etc. These technologies make creating the general application architecture – the server, as well as the client-side, much easier.

Let's move on to realizing our web application.

Create a **new HTML file** in VS Code and name it sumator.html.

A screenshot of a computer

Description automatically generated

Then write the following code:

A screenshot of a computer program

Description automatically generated

This code **creates one web form containing three text fields and a button**. It's specified that pressing the **[Calculate]** button will call the calculate action.

Let's examine the code we just wrote.

A screenshot of a computer code

Description automatically generated

On the first 2 rows, we declare that we're describing an **HTML page**, using the <html> and <body> tags, the latter declaring the start of our page's body – it's the main part that will be visualized. Respectively, the last 2 rows represent the corresponding **closing tags**, which declare the end of that particular part of our HTML page. Look more closely and you'll notice they're different from the **opening tags** – they have a forward slash '/' before the tag name - </body>, for example.

Using HTML tags in our page's body we'll describe what we want to be visualized on our webpage – three fields where numbers can be entered. This can be achieved with the <input type ="number"> tag. Using this tag we declare that we want the input of type number to be visualized. The additional **attribute** – id, serves to let us determine a unique name for this **tag**. What we decide to call it is completely up to us. We'll need the id attribute later to get hold of information about this particular **HTML element**.

Then, on the 7th row we have another input tag declared, but this time it's of type button. This is how we indicate we want an element that, upon being clicked, produces some result.

Let's now also look at the **JavaScript code** we wrote:

A computer screen shot of a code

Description automatically generated

First, we **declare the calculate() function** which reads the information from our first two text fields (from our HTML page), then calculates their sum and **assigns** it as the third field's value. The function spans from the 9th to the 17th row. We'll learn what functions are, how they're declared and called, in this book's following chapters.

Looking more closely at the **function's body**, we observe that on the 10th row the **variable firstNumber is declared** and then the value of the text field with id=firstNumber is assigned to it. The parseInt(…) function makes sure that the text that's been entered will be converted to a number. Then similarly, we get the value of our second text field (with id=secondNumber). Finally, using the same mechanism, instead of obtaining the value of the third field with id=result, we **assign** a value to it by placing it on the left side of the equation.

Then on the 19th and 20th row we access our button and tell it to **listen** for click events which, when received, will trigger the function calculate() we just **declared**. In other words, when we use our mouse to click the button, the calculate() function will execute. The application is ready. We can start it by finding its location and opening it. It will load inside the browser by default.

Does this seem frightening? **There's no need to be scared!** We have a lot more to learn, to reach the level of knowledge and skills required to write web-based applications with ease like in the example above, as well as much bigger and more complex applications. If it all makes little sense, just keep going without worrying. In time, you will remember with a smile how incomprehensible and exciting your first collision with web programming was. If you're having issues with the example above, **watch the video** at the beginning of this chapter – the application is created step by step and explained in great detail. You can also ask for assistance in the **SoftUni Reddit:** [**https://www.reddit.com/r/softuni/**](https://www.reddit.com/r/softuni/).

The purpose of the example above (web application) is not to teach you, but to allow you to dive a little deeper into programming, **to fuel your interest** in software development, and to inspire you to study hard. **You have a lot more to learn**, but it is interesting, isn't it?

# Chapter 2.1. Simple Calculations

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

* How to work with **data types and variables** which we need when processing numbers and strings.
* How to **print** a result on the screen.
* How to **read** a custom input.
* How to do simple **arithmetic operations**: addition, subtraction, multiplication, division, concatenate strings.
* How to **round** numbers.

## Calculations in Programming

We know that computers are machines that process data. All **data** is stored in the computer memory (RAM) in **variables**. Variables are named memory areas that store data of a certain type, for example, number or string. Each **variable** in JavaScript has a **name** and **value**. Here is how we would define a variable by assigning it a value at the same time as declaring it:

A grey and black text

Description automatically generated

After processing, the data is stored again in variables (i.e. somewhere in the memory set aside by our program).

## Data Types and Variables

In programming each variable stores a certain **value** of a particular **type**. For example, data types can be a **number**, **string** (text), a **boolean** type, **data**, **list**, etc. Here are some examples of data types and values for them:

* **number** - type of number: 1, 42, -5, 3.14, NaN, …
* **string** - type of text (string): 'Hello', "Hi", 'Beer', …
* **boolean** - boolean type: true, false
* **Date** - date: Tue Jul 04 2017, ……

**JavaScript** language has three keywords for declaring a variable - **var**, **const**, and **let**. The main difference between **let** and **var** is in the scope of the variable. We use **const** when we are sure that what we assign to the variable will not change. A little further in the book, we will find out more details about the range of variables but for now, we will use the word **let** to declare a new variable.

## Print a Result on The Screen

For printing text, number, or another result on the screen, it's necessary to call the built-in method **console.log()**. With it we can print both the value of a variable and directly text or number:

console.log(42); // prints number

console.log('Hello!'); // prints string

let msg = 'Hello, JavaScript!';

console.log(msg); // prints a value of variable

## Reading a User Input as an Integer:

For reading a user input as a **number** is necessary to **define an argument** of our function:

function sum([arg1, arg2]) {

let a = parseInt(arg1);

let b = parseInt(arg2);

...

}

Let's note that the arguments **arg1** and **arg2** can be a different data type than the one we want. That's why it's necessary to convert them into a suitable one. If it's not done for the program **each number** will be just a **string** with which **we can't do operations** arithmetic operations.

### Problem: Square Area

For example, let's look at the following function which reads an integer from the console, multiplies it by itself (squares it), and prints the result from the multiplication. That's how we can calculate square area by side length:

function calculateSquareArea([arg1]) {

let a = parseInt(arg1);

let area = a \* a;

console.log('Square area = ' + area);

}

If we call our function with parameter 3 - **calculateSquareArea([3])** the result will be - **Square area = 9**. Here's how our code looks like in action in the web browser's JavaScript console:

A screenshot of a computer

Description automatically generated

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

#### How Does The Example Work?

On the first line with **function calculateSquareArea([arg1]) {** we define our function by giving it a name and setting arguments that it needs. In our case, we have one argument which is the side of the square.

On the next line with **let a = parseInt(arg1);** we get the argument of the function **arg1** and convert it to an integer with the method **parseInt(arg1);**. The result is saved in variable **a**.

**Note**: If **arg1** contains a **floating-point number**, that will be **rounded to an integer**. Converting a floating number to an integer is performed by **removing** all digits after the decimal point. Example: **parseInt(2.3)** = 2, **parseInt(3.8)** = 3

The next command **let area = a \* a;** is saved in a new variable named **area** - the result of the multiplication **a** by **a**.

The next command **console.log('Square area = ' + area);** prints the specified text by placing the calculated face of the square, which we have saved in the variable **area** next to it.

The above program can be simplified a bit, like this:

function calculateSquareArea([a]) {

let area = a \* a;

console.log('Square area = ' + area);

}

The above code will work correctly because when multiplied, the variable a will be converted to a number. When the input is only a single number, the parentheses [] can also be skipped, like this:

function calculateSquareArea(a) {

let area = a \* a;

console.log('Square area = ' + area);

}

The code can be compact even more, like this:

function calculateSquareArea(a) {

console.log('Square area = ' + a \* a);

}

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#0>. Try all four solutions to the problem.

## Reading Floating-Point Numbers

To read user input as **floating-point number** it's necessary again to **define an argument** to our function. The syntax is similar to reading an integer, but here we have to use the function **parseFloat(...)**:

function sum([arg1, arg2]) {

let a = parseFloat(arg1);

let b = parseFloat(arg2);

...

}

### Problem: Inches to Centimeters

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

function convertInchesToCentimeters([arg1]) {

let inches = parseFloat(arg1);

let centimeters = inches \* 2.54;

console.log('Centimeters = ' + centimeters);

}

Let's call the function and make sure that when passing a value in inches, we get the correct result in centimeters:

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

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#1>.

## Reading a Text Input

Same with other data types, to read a **string** it's necessary to **define an argument** to our function and after that assign it to a variable:

function print([arg1]) {

let text = arg1;

...

}

### Problem: Greeting by Name

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

function sayHello([arg1]) {

let name = arg1;

console.log(`Hello, ${name}!`);

}

In this case, the expression **${name}** will be replaced with **the value of the variable name**. If we call the function with the name "Ivan", that will be the result:

sayHello(['Ivan']); // Hello, Ivan!

#### Testing in Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#2>.

## Concatenating Text and Numbers

When printing text, numbers and other data **we can concatenate them** by using templates **`variable = ${variable}`** . In programming these templates are called **placeholders**. Pay attention: We need to use italicized apostrophes ` (**backticks**) instead of normal quotes to recognize the template:

function printInfo([firstName, lastName, age, town]) {

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

}

We call the function with test parameters again and make sure that it works:

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

Except for variables, we can make simple calculations in the templates.

The same variable can be used as a template more than once. Here's an example:

let a = 1;

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

The result is:

1 + 1 = 2

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#3>.

## Arithmetic Operations

Let's examine the basic arithmetic operations in programming.

### Summing Numbers (operator +)

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

let a = 5;

let b = 7;

let sum = a + b; // the result is 12

### Subtracting Numbers (Operator -)

Subtracting numbers is done using the **-** operator:

function substractNumbers([arg1, arg2]) {

let a = parseInt(a);

let b = parseInt(b);

let result = a - b;

console.log(result);

}

Let we check the result of the execution of this program (with numbers 10 and 3):

substractNumbers([10, 3]); // 7

### Multiplying Numbers (Operator \*)

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

let a = 5;

let b = 7;

let product = a \* b; // 35

### Dividing Numbers (Operator /)

Dividing numbers is done using the **/** operator.

**Note:** Float numbers **divided by 0** do not cause an exception and the result is **+/- infinity** or the special value **Infinity**.

Here are a few examples with the division operator:

console.log(10 / 2.5); // Result: 4

console.log(10 / 4); // Result: 2.5

console.log(10 / 6); // Result: 1.6666666666666667

console.log(a / 0); // Result: Infinity

console.log(-a / 0); // Result: -Infinity

console.log(0 / 0); // Result: NaN (Not a Number), i.e. the result

// The operation hasn't a valid numeric value

## Concatenating Text and Numbers

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

let firstName = "Maria";

let lastName = "Ivanova";

let age = 19;

let str = firstName + " " + lastName + " @ " + age;

console.log(str); // Maria Ivanova @ 19

Here is another example:

let a = 1.5;

let b = 2.5;

let sum = "The sum is: " + a + b;

console.log(sum); // The sum is: 1.52.5

Do you notice anything strange? Maybe you expected the numbers **a** and **b** to sum? The concatenation works from left to right and the above result is correct. If we want to sum the numbers, we will have to use **brackets** to change the order of operations:

let a = 1.5;

let b = 2.5;

let sum = "The sum is: " + (a + b);

console.log(sum); // The sum is: 4

## Numerical Expressions

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

let expr = (3 + 5) \* (4 – 2);

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

### Problem: Concatenate Data

Write a function, that receives a first name, last name, age and city and prints a message of the following kind:

You are <firstName> <lastName>, a <age>-years old person from <town>.

### Hints and Guidelines

We write a function that receives the input in the **following order**:

A close up of a computer screen

Description automatically generated

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

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

Next, the solution should be tested locally using **[Ctrl+F5]**.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#3>.

### Problem: Trapeziod Area

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

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);

}

Because we want our function to work with both integers and floating numbers, we use **parseFloat()**. If we start the function and enter values for sides: **3**, **4**, and **5**, we will obtain the following result::

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

#### Testing in The Judge System

Test your solution here:

<https://judge.softuni.org/Contests/Practice/Index/927#4>.

## Rounding Numbers

Sometimes when we work with floating numbers, it's necessary to bring them to integers. This bringing is named **rounding**. The programming language **JavaScript** provides us with several methods for rounding numbers:

* **Math.ceil(…)** - **rounding up** to next (greater) integer:
* let up = Math.ceil(45.15); // up = 46
* **Math.floor(…)** - **rounding down** to previous (less) integer:
* let down = Math.floor(45.67); // down = 45
* **Math.trunc(…)** - **cutting** the decimal places:
* let trunc = Math.trunc(45.67); // trunc = 45
* **Math.round(…)** - rounding is done as a **basic rule for rounding numbers** - if the decimal part is less than 5, rounding is to the previous number and if it's greater than 5 - to the next:
* Math.round(5.439); // 5
* Math.round(5.539); // 6
* **.toFixed([number of characters after the decimal point])** - rounding to **the closest** number:
* (123.456).toFixed(2); // 123.46
* (123).toFixed(2); // 123.00
* (123.456).toFixed(0); // 123
* (123.512).toFixed(0); // 124

### Problem: Circle Area and Perimeter

Let's write a function that receives an input of **the radius r** of a circle and **calculates the area and the perimeter** of the circle.

Formulas:

* Area = π \* r \* r
* Perimeter = 2 \* π \* r
* π ≈ 3.14159265358979323846…

function calculateCircleAreaAndPerimeter([arg1]) {

let r = parseInt(arg1);

console.log("Area = " + Math.PI \* r \* r);

// Math.PI - Built-in JavaScript constant for the value of the number π

console.log("Perimeter = " + 2 \* Math.PI \* r);

}

Let's call the function with **radius r = 10**:

calculateCircleAreaAndPerimeter([10])

The result is:

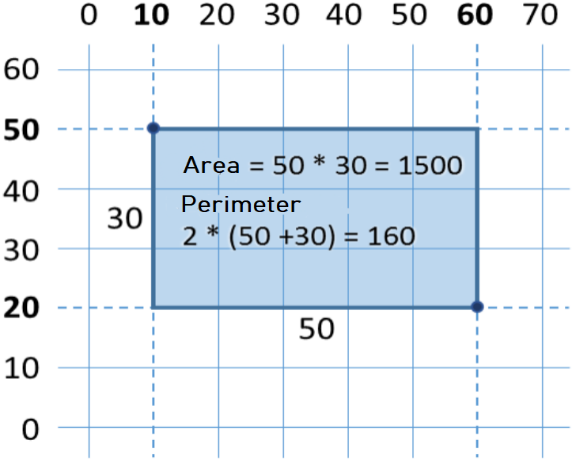


#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#5>.

### Problem: 2D Rectangle Area

The rectangle is given with the **coordinates of two of its opposite angles**. Calculate its **area and perimeter** :



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

function calculateRectangleArea([arg1, arg2, arg3, arg4]) {

let x1 = parseFloat(arg1);

let y1 = parseFloat(arg2);

let x2 = parseFloat(arg3);

let y2 = parseFloat(arg4);

// Calculating the sides of the rectangle:

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));

}

We use the method **Math.max(x1, x2)** to find the higher value from **x1** and **x2** and identically **Math.min(y1, y2)** to find the lower of both values.

Let we call the function with testing values from the coordinate system:

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

// 160

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#6>.

## Exercises: Simple Calculations

Let's strengthen the knowledge gained throughout this chapter with a few more exercises.

### Empty **JS** File for our Solution of the Problem in Visual Studio Code

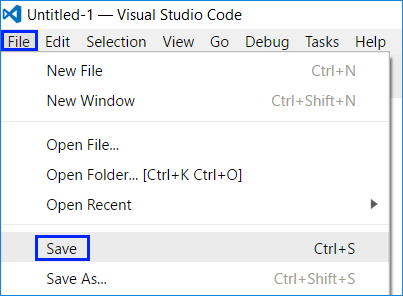
We start by creating an empty **JS file** in Visual Studio Code. In the current practical exercise we'll create and add a new **JS file** for each task, to organize the solutions of the tasks from the exercises:

We start Visual Studio Code and create a **New File:** [**File**] -> [**New File**]:

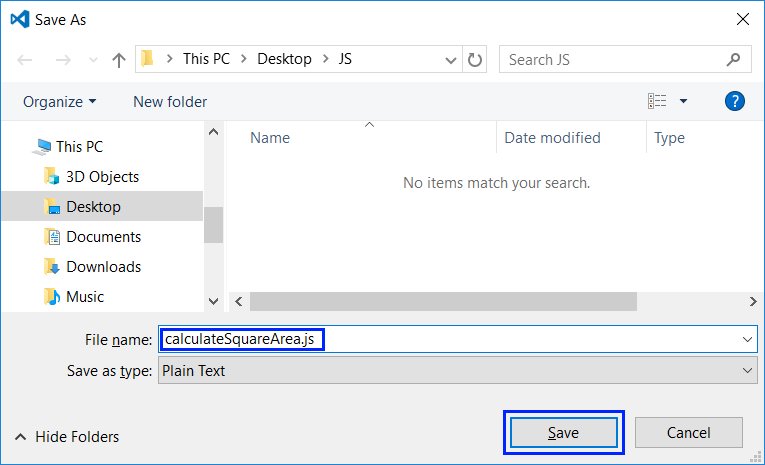
A screenshot of a computer

Description automatically generated

**Save** the file from [**File**] -> [**Save**] or by keyboard shortcut [**Ctrl + S**]:



Enter a **significant name** and expansion **.js** to our file, then press the button [**Save**]:



### Problem: Triangle Area

Write a function that receives arguments which are **a side and a height of a triangle** and calculates its area. Use **the formula** for triangle area: **area = a \* h / 2**. Round the result to **2 digits after the decimal point using area.toFixed(2)**.

#### Sample Input and Output

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

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#7>.

### Problem: Celsius to Fahrenheit

Write a function that reads **degrees on the Celsius scale** (°C) and converts them to **degrees on the Fahrenheit scale** (°F). Look on the Internet for a proper [formula](https://bfy.tw/3rGh) to do the calculations. Round the result to **2 digits after the decimal point**. Here are a few examples:

#### Sample Input and Output

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

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#8>.

### Problem: USD to BGN

Write a function that reads **an angle in** [**radians**](https://en.wikipedia.org/wiki/Radian) (**rad**) and converts it in [**degrees**](https://en.wikipedia.org/wiki/Degree_(angle)**)** (deg). Look for a proper formula on the Internet. The number **π** in **JavaScript** programs is available through **Math.PI**. Round the result to the nearest integer using the **Math.round(…)** method.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 3.1416 | 180 |
| 6.2832 | 360 |
| 0.7854 | 45 |
| 0.5236 | 30 |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#9>.

### Problem: Converter - USD to BGN

Write a function for **conversion of US dollars** (USD) **into Bulgarian levs** (BGN). **Round** the result to **2 digits** after the decimal point. Use a fixed rate between a dollar and lev: **1 USD = 1.79549 BGN**.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 20 | 35.91 BGN |
| 100 | 179.55 BGN |
| 12.5 | 22.44 BGN |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#10>.

### Problem: \* Currency Converter

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

| **Rate** | **USD** | **EUR** | **GBP** |
| --- | --- | --- | --- |
| 1 BGN | 1.79549 | 1.95583 | 2.53405 |

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

#### Sample Input and Output

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

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#11>.

### Problem: \*\* 1000 Days After Birth

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

#### Sample Input and Output

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

#### Hints and Guidelines

* Look for information about the data type **Date** in JavaScript and in particular look at the methods **setDate(...)**, **getDate()**, **getMonth()** and **getYear()**. With their help, you can solve the problem without the need to calculate days, months, and leap years.
* **Don't print** anything additional on the console except for the wanted date!

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/927#12>.

## Graphic Applications with Numerical Expressions

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

### Web Application: \*\*\* Converter - BGN to EUR!

Create a web application that calculates the value in **Euro** (EUR) of the monetary amount given in **Bulgarian levs** (BGN). By changing the amount in BGN, the amount in EUR has to be recalculated automatically. Use the fixed-rate BGN / Euro: **1.95583**.



Similarly, as in the first chapter ("First Steps in Programming"), we will use the languages **JavaScript**, **HTML**, and **CSS** for our application.

1. The first step is to **create a folder** in which we will store all the files that are needed for our application.
2. Then we need to create an HTML file in the folder: **index.html**

<!DOCTYPE html>

<html>

<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>

Note that each HTML page must have a **specific structure**. For example, always the main code we write is in the tag **<body>**, and always the title of the page is in the tag **<title>**.

1. We have the structure of the page, it remains to add a **JavaScript** file with the logic itself. We create a new file and name it **converter.js**

function eurConverter() {

let bgn = document.getElementById("bgn").value;

let eur = (bgn / 1.95583).toFixed(2);

document.getElementById("euro").value = eur;

}

1. Once we have the logic of the application, we need to find a way to tell where to use it. To do this we need to make 2 changes to the existing **index.html** file:

First, we add the following line just below the **title** tag, through which the connection between the files is made **index.html** and **converter.js**:

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

And second, we find and replace the **input** field with the type **button** with the following code. In this way we set **when clicking** on the button [**Convert!**] to call the function **eurConverter()**:

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

If we start the file **index.html** from the folder, we should have a working application that converts from BGN to EUR:



Let's make it more beautiful.

1. We create a new file with extension **\*.css** and name **index**. [CSS](https://www.w3schools.com/html/html_css.asp) is used to stylize the elements in HTML. We open the file **index.html** and add this line in the tag **<head>**:

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

In the file **index.css** we write the following code (we define styles for the individual elements of the HTML format):

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;

}

1. Start the file **index.html**:

A screenshot of a computer

Description automatically generated

### Web Application: \*\*\* Catch The Mouse!

Upon moving the mouse cursor onto the button, it moves to a random position. This way it creates the impression that **the image runs from the mouse** and it is hard to catch“. When the image gets “caught”, a congratulations message is shown.

**Hint**: Write an Event Handler **mouseover** and move the image to a random position. Use the random numbers generator **Math.random()**. The position of the image is set from the property **style.position**. To "catch the mouse" **onclick** to "catch the mouse".

A computer screen shot of a mouse

Description automatically generated

1. We create a new folder **catch-the-mouse** in which we will save the files for the application.
2. We create two files in the folder: **index.html** and **app.js**. The folder structure must look like this:

A screenshot of a computer

Description automatically generated

1. You can help yourself with the code below:

The file **index.html** must look like this:

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<title>Catch the mouse!</title>

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

</head>

<body>

<img id="image" src="images/mouse.jpg" />

</body>

</html>

The file **app.js** must look like this:

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!")

}

1. We find an image from the internet and add it by naming it **mouse.jpg**.

Test the application by opening the project folder in **explorer** and launching the file **index.html**:

A screenshot of a computer

Description automatically generated

1. Done the application.

If you have any difficulties, ask in **the Softuni Reddit**: <https://www.reddit.com/r/softuni/>.

## What Have We Learned from This Chapter?

Let's summarize what we learned from this chapter of the book:

* **Reading an user input**: **function sum([number1, number2])**.
* **Converting to number**: **let num = parseInt(arg1)**, **let num = parseFloat(arg1)**.
* **Aritmetic operations** and using the relevant **aritmetic operators** [+, -, \*, /, ()]: **let sum = 5 + 3**.
* **Print text by using concatenation**: **console.log(`3 + 5 = ${3 + 5}`)**.
* The different types of **rounding** numbers: **Math.ceil()**, **Math.trunc()**, **Math.floor()** and **.toFixed()**

# Chapter 2.2. Simple Calculations – Exam Problems

In the previous chapter, we explained how to **pass numbers** to functions and how to **print the output** in the console. We went through the main arithmetical operations and briefly mentioned data types. In this chapter, we are going to practice what we have learned by solving a few **more complex exam problems**.

## Reading Numbers

Before going to the tasks, we are going to revise the most important aspects of what we have studied in the previous chapter. We will start by creating a function, which will read a number.

### Reading an Integer

We need to create a function, which will get one argument **arg1**. We will create a variable in the function, where we will store the number (for example **num**), combined with the method **parseInt(…)**, which converts a string to an integer:

function readNumber(arg1) {

let num = parseInt(arg1);

}

### Reading a Floating-Point Number

The same way we read an integer one, but this time we use the method **parseFloat(…)**:

let num = parseFloat(arg1);

## Printing Text Using Placeholders (placeholder)

A **placeholder** is an expression that is replaced with a particular value while printing an output. For this placeholder to work, we will have to use backticks **`…`** . The method **console.log(…)** supports printing a string based on a placeholder, where the first arguments, which have to be printed are written in the **${…}**:

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.

## Arithmetic Operators

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

### Operator +

let result = 3 + 5; // the result is 8

### Operator -

let result = 3 - 5; // the result is -2

### Operator \*

let result = 3 \* 5; // the result is 15

### Operator /

let result2 = 5 / 2; // the result is 2.5 (floating-point division)

## Concatenation

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

let firstName = "Ivan";

let lastName = "Ivanov";

let age = 19;

let str = firstName + " " + lastName + " is " + age + " years old";

// Ivan Ivanov is 19 years old

## Exam Problems

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

## Problem: Training Lab

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

### Input Data

The program reads **two numbers** (arguments), one per line: **l** (length in meters) and **w** (width in meters).

Constraints: **3 ≤ w ≤ l ≤ 100**.

### Output Data

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

### Sample Input and Output

| **Input** | **Output** | **Figure** |
| --- | --- | --- |
| 15 8.9 | 129 |  |
| 8.4 5.2 | 39 |  |

#### Clarification of the Examples

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

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

### Hints and Guidelines

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

#### Idea for Solution

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

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

#### Choosing Data Types

From the example, we see that a real number with whole and fractional part can be given as an input, but in **JavaScript**, there is only one primitive type of number (**Number**), therefore, this will not be a problem. Choosing a data type for the next variables depends on the method we choose to solve the problem. As with any programming task, this one has **more than one way to be solved**.

#### Solution

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

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

The first thing we have to do is read the input from the console. We create a function, which will get two arguments. Then we need to save their values in two variables by using the method **parseFloat(…)** to convert the string (text) value into a fractional number:

A black text on a white background

Description automatically generated

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

|  |  |
| --- | --- |
| A yellow triangle with a exclamation mark  Description automatically generated | **Search in Google!** Whenever we have an idea how to solve a particular problem, but we do not know how to write it in JavaScript or we are dealing with one that many other people have had before us, the easiest way to solve it is by looking for information on the Internet. |

In this case, we can try with the following search: "[**JavaScript get whole number part of double**](https://www.google.com/?q=JavaScript+get+whole+number+part+of+double)". One possible way is to use the method **Math.trunc(…)**. The code down below is blurred on purpose and it should be completed by the reader:

A black and white text on a white surface

Description automatically generated

With **console.log(…)** we print the result in the console:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/928#0>.

## Problem: Vegetable Market

A gardener is selling his harvest on the vegetable market. He is selling **vegetables for N BGN per kilogram** and **fruits for M BGN per kilogram**. Write a program that **calculates the earnings of the harvest in Euro**. (Assume that **1 EUR** is equal to **1.94 BGN**).

### Input Data

**Four arguments** are passed to the function:

* Vegetable price per kilogram – a floating-point number.
* Fruit price per kilogram – a floating-point number.
* Total kilograms of vegetables – an integer.
* Total kilograms of fruits – an integer.

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

### Output Data

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

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 0.194 19.4 10 10 | 101 |

**Clarification for the first example:**

* Vegetables cost: 0.194 BGN. \* 10 kg. = **1.94 BGN.**
* Fruits cost: 19.4 BGN. \* 10 kg. = **194 BGN.**
* Total: **195.94 BGN. = 101 EUR**.

| **Input** | **Output** |
| --- | --- |
| 1.5 2.5 10 10 | 20.6185567010309 |

### Hints and Guidelines

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

#### Idea for Solution

Let's first go through the problem requirements. In this case, we have to calculate **the total income** from the harvest. It equals **the sum of the earnings from the fruits and vegetables** which we can calculate by multiplying **the price per kilogram by the quantity**. The input is given in BGN and the output should be in euros. It is assumed that 1 EUR equals 1.94 BGN, therefore, to get the wanted **output value, we have to divide the sum by 1.94**.

#### Choosing Data Types

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

#### Solution

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

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

To read the input, we declare variables, which we have to name carefully so that they can give us a hint about the values they store. With the methods **parseInt(…)** and **parseFloat(…)**, we convert the particular string value into int and double.

A black and white text

Description automatically generated

We do the necessary calculations:



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



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/928#1>.

## Problem: Change Tiles

On the ground in front of an apartment building **tiles need to be placed**. The ground has a **square shape with a side of N meters**. The tiles are **"W" meters wide** and **"L" meters long**. There is one bench on the ground with a **width of "M" meters and a length of "O" meters**. There is no need to place tiles under it. Each tile is replaced for **0.2 minutes**.

Write a program that **reads from the console the size** of **the ground, the tiles, and the bench**, and calculates **how many tiles are needed** to cover the ground and what is the total **time for placing all of the tiles**.

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

### Input Data

**5 arguments** are passed to the function:

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

### Output Data

Print on the console **two numbers**:

* **number of tiles** needed for the repair
* **total time for placing them**

Each number should be on a new line and rounded to the second decimal place.

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 20 5 4 1 2 | 19.9 3.98 |

**Explanation of the example:**

* **Total area** = 20 \* 20 = 400.
* **Area of the bench** = 1 \* 2 = 2.
* **Area for covering** = 400 – 2 = 398.
* **Area of tiles** = 5 \* 4 = 20.
* **Needed tiles** = 398 \/ 20 = 19.9.
* **Needed time** = 19.9 \* 0.2 = 3.98.

| **Input** | **Output** |
| --- | --- |
| 40 0.8 0.6 3 5 | 3302.08 660.42 |

### Hints and Guidelines

Let's make a draft to clarify the task requirements. It can look the following way:

A graph with a bench and a bench

Description automatically generated

#### Idea for Solution

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

We calculate the area of a single tile by **multiplying its two sides with one another** **W \* L**. As we already stated, now we have to **divide the area for covering by the area of a single tile**. This way, we find the number of necessary tiles. We multiply it by **0.2** (the time needed for changing a tile). Now, we have the wanted output.

#### Choosing Data Types

The length of the side of the ground, the width, and the length of the bench will be given as **integers**, therefore, to store their values, we can declare **variables converted by the method parseInt(…)**. We will be given floating-point numbers for the width and the length of the tiles and this is why we will use **parseFloat(…)**.

#### Solution

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

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

The first thing we have to do is go through the **input data** of the task. It is important to pay attention to the sequence they are given in. We create the necessary variable where we will store the input data and with the methods **parseInt(…)** and **parseFloat(…)** we convert the particular string value into an int or double.

A computer code with text

Description automatically generated

After we have initialized the variables and have stored the corresponding values in them, we move to the **calculations**. The code below is blurred on purpose, so the reader can think by himself over it:



**We calculate the values** that we have to print on the console. **The number** of necessary **tiles** is obtained by **dividing the area** that needs to be covered **by the area of a tile**.

In the task is specified that the number of the output should be rounded **to the second decimal place**. That is why we cannot just print the value with **console.log(…)**. We will use the method **Math.round(…)**, which is rounding to the closest integer number. To round it to the second decimal place, we will use the method after the number is multiplied by 100 and then we will divide the result by 100:

A close up of text

Description automatically generated

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/928#2>.

## Problem: Money

Some time ago, **Pesho bought Bitcoins**. Now, he is going on vacation in Europe **and he needs euros**. Apart from Bitcoins, he has **Chinese yuans** as well. Pesho wants to **exchange his money for euros** for the tour. Write a program that **calculates how much euro he can buy, depending on the following exchange rates**:

* **1 Bitcoin = 1168 BGN.**
* **1 Chinese yuan (CNY) = 0.15 USD.**
* **1 USD = 1.76 BGN.**
* **1 EUR = 1.95 BGN.**

The exchange office has a **commission fee of 0% to 5% from the final sum in euro**.

### Input Data

**3 arguments** are passed to the function:

* **Number of Bitcoins** - integer within the range of [**0 … 20**].
* **Several Chinese yuans** - a floating-point number within the range of [**0.00 … 50 000.00**].
* **Commission fee** - a floating-point number within the range of [**0.00 … 5.00**].

### Output Data

Print one number on the console – **the result of the exchange of currencies**. The output should be rounded **to the second decimal place**.

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 1 5 5 | 569.67 |

**Explanation**:

* 1 Bitcoin = 1168 BGN
* 5 Chinese yuan (CNY) = 0.75 USD
* 0.75 USD = 1.32 BGN
* **1168 + 1.32 = 1169.32 BGN = 599.651282051282 EUR**
* **Commission fee:** 5% of 599.651282051282 = **29.9825641025641**
* **Result**: 599.651282051282 - 29.9825641025641 = **569.668717948718 EUR**

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 20 5678 2.4 | 12442.24 | 7 50200.12 3 | 10659.47 |

### Hints and Guidelines

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

#### Idea for Solution

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

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

Only the final step is left: **calculating the commission fee** and subtracting the new sum from the total one. We will obtain a **floating-point number** for the commission fee, which will be a particular **percent of the total sum**. Let's divide it by 100, to calculate its **percentage value**. Then we will multiply it by the sum in euro and divide the result from the same sum. Print the final sum on the console.

#### Choosing Data Types

**Bitcoins** are given as an **integer**, therefore, we can declare a variable converted by the method **parseInt(…)**. For Chinese yuan and commission fee, we obtain a **floating-point number**, therefore, we are going to use **parseFloat(…)**.

#### Solution

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

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

**We declare the variables** that we are going to use and again we have to choose **meaningful names**, which are going to give us hints about the values they store. We initialize their values: we create variables, where we will store the string arguments passed to the function and convert them to int or double:

A black text on a white background

Description automatically generated

We do the necessary calculations:

A black and white text

Description automatically generated with medium confidence

A green and white text

Description automatically generated with medium confidence

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

Finally, we have to print the result in the console. We should notice that we have to format the output **to the second decimal place**. The difference between this and the previous task is that here, even if the number is an integer, **we have to print it to the second decimal place** (for example **5.00**). To do so, we can use the method **toFixed(…)**. By using it, we can covert a number into a, saving the specified numbers after the decimal point.



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

A close up of a number

Description automatically generated

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

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/928#3>.

## Problem: Daily Earnings

Ivan is a programmer in an American company, and he **works** at home **approximately N days per month** by earning **approximately M USD per day**. At the end of the year, Ivan **gets a bonus**, which **equals 2.5 of his monthly salary**. In addition, **25% of his annual salary goes for taxes**. Write a program that **calculates what is the amount of Ivan's net average earnings** in BGN per day, as he spends them in Bulgaria. It is accepted that one year has exactly 365 days. The exchange rate of US USD to BGN will be passed to a function.

### Input Data

**3 arguments** are passed to the function:

* **Workdays per month** - an integer within the range of [**5 … 30**].
* **Daily-earnings** - A floating-point number within the range of [**10.00 … 2000.00**].
* **Exchange rate of USD to EUR** /1 USD = X BGN/ - A floating-point number within the range of [**0.99 … 1.99**].

### Output Data

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

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 21 75.00 1.59 | 74.61 |

**Explanation**:

* **One monthly salary** = 21 \* 75 = 1575 USD.
* **Annual income** = 1575 \* 12 + 1575 \* 2.5 = 22837.5 USD.
* **Taxes** = 25% of 22837.5 = 5709.375 USD.
* **Net annual income in USD** = 17128.125 USD = 27233.71875 BGN.
* **Average earnings per day** = 27233.71875 / 365 = 74.61 BGN.

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 15 105 1.71 | 80.24 | 22 199.99 1.50 | 196.63 |

### Hints and Guidelines

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

#### Idea for Solution

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

#### Choosing Data Types

**The working days per month** are given as an integer, therefore, we can declare a variable by using **parseInt(…)** to store their value as a number. For both the **earned money** and the **exchange rate of USD to BGN**, we will obtain a floating-point number, therefore, we will use **parseFloat(…)**.

#### Solution

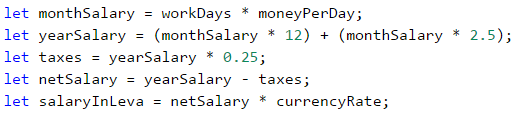
Again: after we have an idea of how to solve the problem and we have considered the data types that we are going to use, we can start **writing the program**. As in the previous tasks, we can divide the solution into three smaller tasks:

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

We **declare the variables** that we are going to use by trying to choose **meaningful names**. We create a variable to store the arguments passed to the function, by converting the string to integer or floating number by using **parseInt(…)/parseFloat(…)**:



We do the calculations:



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

Finally, we have to print the result on the console. We notice that the number has to be rounded up to the second digit after the decimal point. To do that, we can use **.toFixed(…)** just like the previous task:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/928#4>.

# Chapter 3.1. Simple Conditions

In the present chapter, we will take a look at the **conditional constructs in the JavaScript programming language**. By implementing these constructs, our program can produce a different output based on a given specific input. We will explain the syntax of the conditional operators (**if** and **if-else**) by implementing appropriate examples and also we will take a look at the range in which a variable lives (its **scope**). Finally, we will go over different **debugging** techniques, to follow the programming steps through which our program goes during its run.

## Conditional Operators

In programming, we can compare values through the use of the following **operators**:

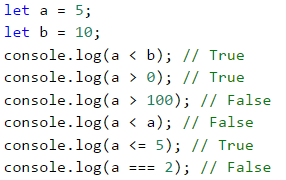
* operator **<** (less than)
* operator **>** (greater than)
* operator **<=** (less than or equals)
* operator **>=** (greater than or equals)
* operator **===** (equals)
* operator **!==** (not equal; different than)

The result from a comparison is the so-called Boolean value, which can be either **true** or **false** depending on the evaluated result being either true or false.

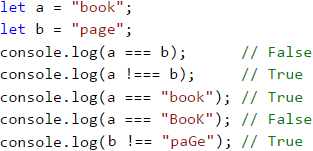
It is important to note that in **JavaScript** there are further comparison operators - for **comparison** **==** and **difference** **!=**. The implementation of these operators without having intimate knowledge of their evaluation may lead to unexpected results and problems, for the moment we will not take look at them in detail.

Additional information on the differences between the two types of comparison operators can be found on the following link: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Comparison_Operators>

### Problems of Number Comparisons



### Problems of comparing "text" (string) type variables



It is important to note that the case of the letters **upper case** or **lower case** is important for the comparison. If the compared values are not **completely identical**, the output will always be **false**.

### Comparison Operators

In **JavaScript** we can use the following operators to compare data:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Notation** | **Applicable for** |
| Equals | === | numbers, strings, dates |
| Not equal | !== |
| Greater than | > | numbers, dates, other comparable data types |
| Greater than or equal | >= |
| Less than | < |
| Less than or equal | <= |

## Simple if Comparisons

In programming, we often **check particular conditions** and perform various actions depending on the result of the comparison. This is done through the **if** comparison, which has the following structure:

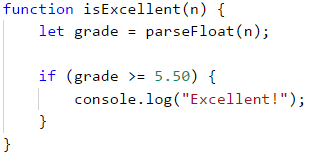
if (Boolean condition) {

// body of the conditional construct;

}

### Problem: Excellent Result

We take the grade as an input argument to our function and upon evaluation, we check if the input value is an excellent grade (**≥ 5.50**).



Test the example code locally. Try entering different grades, for example, **4.75**, **5.49**, **5.50**, and **6.00**. For grades **less than 5.50** the program will not give any output, however for grades of **5.50 or greater**, the output will be "**Excellent!**". The function is called by simply writing its name and filling the input value in the parenthesis:



#### Testing in The Judge System

You can test the solution example here: <https://judge.softuni.org/Contests/Practice/Index/929#0>.

## If-Else Conditional Constructs

The **if** conditional can also have an **else** option to provide a specific action to be performed in case the Boolean expression (which is specified at the beginning **if (Boolean expression)**) returns a negative/falsy result (**false**). Written in this way the **conditional statement** is called **if-else** and its behavior is as follows: if the result of the condition is **positive / truthy** (**true**) - a set of instructions is executed. By contrast, when the result is **negative / falsy** (**false**) - a different set is executed. The format of this structure in **JavaScript** is as follows:

if (Boolean condition) {

// Condition body to be executed if a condition is true

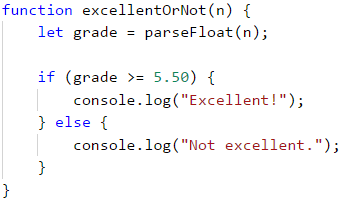
} else {

// else structure body to be executed if a condition is false

}

### Problem: Excellent or Not

Similarly to the example above, we input a grade and check if it is excellent, but this time we should **output a result in both cases**:



#### Testing in Judge System

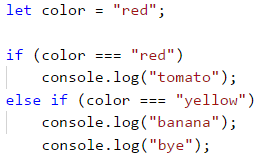
You can test your solution at the following link: <https://judge.softuni.org/Contests/Practice/Index/929#1>.

## About The Curly Braces { } After an If / Else

When we have **only one command** in the body of the **if statement**, we can **skip the curly braces**, indicating the body of the conditional operator. When we need to execute a **block of code** (group of commands), curly braces are **mandatory**. In case the braces are omitted, **only the first line of code** will be executed after the **if statement**.

|  |  |
| --- | --- |
|  | It is a good practice to **always include curly braces** since this makes the code more readable, neater, and cleaner. |

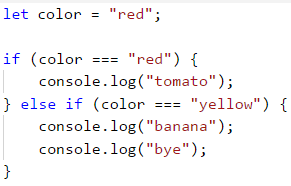
Here is an example, where omitting the curly braces leads to confusion:



Executing the code above will produce the following console output:



With curly braces:



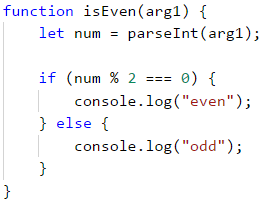
The following output will be printed on the console:



### Problem: Even or Odd

Write a function that checks whether a given input number is **even** or **odd**.

The problem can be solved with a single **if-else** structure and the operator **%**, which returns the **division remainder** of two numbers.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#2>.

### Problem: Greater Number

Write a program that reads two integers and outputs the greater one.

Our first task is to **read** the two numbers. After which through the use of a simple **if-else** structure, in combination with the **greater than operator** (**>**), to perform the comparison. We have deliberately blurred parts of the code so that the reader can implement the learned so far.



#### Testing in The Judge System

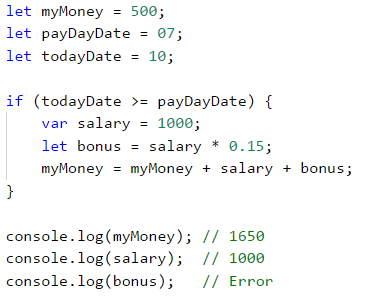
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#3>.

## The Lifetime of a Variable

Every variable has a scope in which it exists, called **variable scope**. This scope specifies where the variable can be used and accessed. In **JavaScript**, there are **two ways** to initialize variables. This is done through the keyword **var** or **let**. It is important to note the difference between them, as to avoid unexpected and unwanted results during the design and execution of our functions.

Variables initialized with the keyword **var** have the properties of **global variables**. They are characterized by the fact that they **can be accessed anywhere, regardless of the location in our code, where they are declared**. Using the keyword **let**, our variable assumes the properties of **a local variable**. This means that its lifetime begins at the row in which it is **defined** and ends until the first closing curly brace **}** (of the function, of the **if statement**, etc.). Owing to this it is important to know that every variable initialized with the keyword **let** within the body of an **if**, **will not be accessible outside its code block / scope**, unless we have defined it higher in the code.

In the example below, in the last lines, we will try to access the defined variables. We will print **myMoney** on the console, because it is declared at the beginning of our function, before the **if structure**, which makes it **available anywhere in the function body**. Even though the **salary** is declared in the **if** structure block, we can print it because it has the properties of a **global variable** (since it is declared with **var**) and can be **used anywhere**. When we try to print the **bonus** variable, which is initialized in the **if structure**, we will get an **error**, since the lifetime of this variable ends with the first closing curly brace **}**, which in this case is the one closing the **if** structure:



Using the keyword **var** was the only way **in the past** to declare a variable, nowadays however this is **not recommended**. Because of this in all examples in this book, we will use the keyword **let**.

It is important to note that there is a **third way** to initialize variables - through the use of the keyword **const**. These variables have the same scope as if defined through **let**, however have one key difference - they are **constant variables**. This means that after the initial assignment, their value is **impossible to be changed or the variable re-defined**.

## Conditional Chaining

Sometimes we have to do a series of checks, before deciding what actions our program will execute. In such cases we can apply the structure **if-else if…-else in series**. For this purpose we employ the following structure:

if (first condition) {

// condition body;

} else if (second condition) {

// condition body;

} else if (third condition) {

// condition body;

}

…

else {

// else structure body;

}

### Problem: Number 0...9 to Text

Print the digits one through nine in English on the console (the numbers are passed as arguments of the function upon call). We can take the digit and through a **series of conditions** print the corresponding English word on the console:

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");

}

}

The program logic of the above example **sequentially compares** the input number with the digits from 1 to 9 **with each consecutive comparison being performed only in case the previous result is not true**. Eventually, if none of the **if** conditionals are satisfied, the last **else clause** is executed.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#4>.

## Exercise: Simple Conditions

To practice the implementation of the conditional constructs **if** and **if-else**, we will take a look at a few practical problems.

### Problem: Bonus Score

We are given an **integer** – several points. Additional **bonus points** are awarded as per the rules described below. Write a function that calculates the **bonus points** for the given number and outputs the **total points** including the bonus.

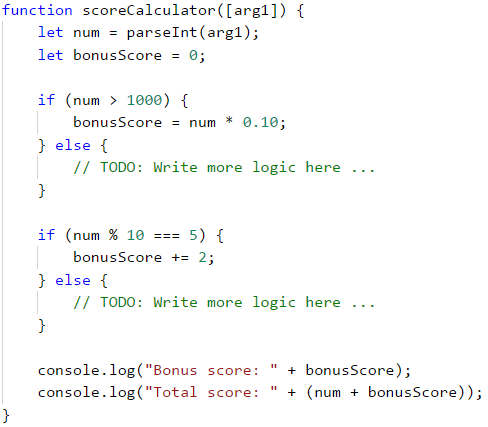
* If the number is **up to 100** inclusive, the bonus points are 5.
* If the number is **larger than 100**, the bonus points are **20%** of the number.
* If the number is **larger than 1000**, the bonus points are **10%** of the number.
* Additional points are awarded as below (added separately from the described above):
  + For **even** numbers -> + 1 p.
  + For numbers, **ending with 5** -> + 2 p.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 20 | 6 26 |
| 175 | 37 212 |
| 2703 | 270.3 2973.3 |
| 15875 | 1589.5 17464.5 |

#### Hints and Guidelines

We can calculate the base and additional bonus score with a series of **if-else-if-else** statements. for the **main bonus points we have 3 cases** (the input is less than or equal to 100, it is between 100 and 1000, and finally it is greater than 1000), for the **additional bonus sore - further 2 cases** (whether the number is even or odd and whether the remainder of division by 5 is 5).



This is an example output when the function is called with 175:



Please note that for this problem the Judge system is set up to ignore any non-number outputs, so we may print explanations along with the number output.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#5>.

### Problem: Sum Seconds

Three athletes finish with some **number of seconds** (between **1** and **50**). Write a function that takes the times of the contestants and calculates their **combined time** in **minutes:seconds** format. Seconds are to be printed with a **leading zero** (2 -> "02", 7 -> "07", 35 -> "35").

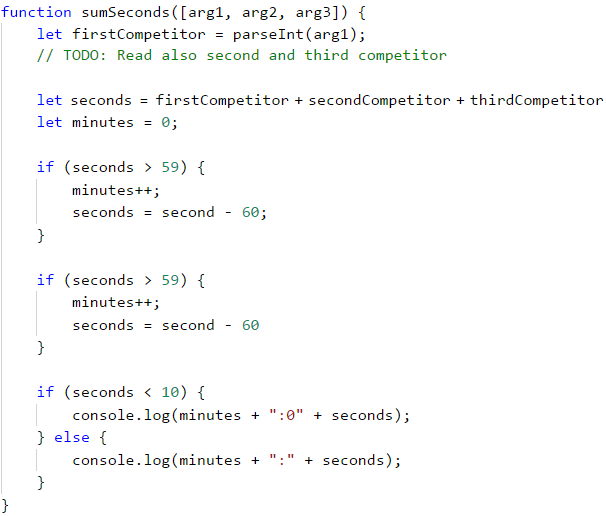
#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 35 45 44 | 2:04 |
| 22 7 34 | 1:03 |
| 50 50 49 | 2:29 |
| 14 12 10 | 0:36 |

#### Hints and Guidelines

Firstly we sum the three numbers, to obtain the seconds total. As we know that **1 minute = 60 seconds**, we should calculate the minutes and seconds in the range 0 to 59:

* If the result is between 0 and 59, we print 0 minutes + calculated seconds.
* If the result is between 60 and 119, we print 1 minute + calculated seconds minus 60.
* If the result is between 120 and 179, we print 2 minutes + calculated seconds minus 120.
* If the seconds are less than 10, we print the number with a leading zero.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#6>.

### Problem: Metric Converter

Write a function, that **converts distance** between the following **8 units of measure**: **m, mm, cm, mi, in, km, ft, yd**. You may use the conversion table below:

| **Input measure** | **Output measure** |
| --- | --- |
| 1 meter (m) | 1000 millimeters (mm) |
| 1 meter (m) | 100 centimetres (cm) |
| 1 meter (m) | 0.000621371192 miles (mi) |
| 1 meter (m) | 39.3700787 inches (in) |
| 1 meter (m) | 0.001 kilometres (km) |
| 1 meter (m) | 3.2808399 feet (ft) |
| 1 meter (m) | 1.0936133 yards (yd) |

The input will be three parameters:

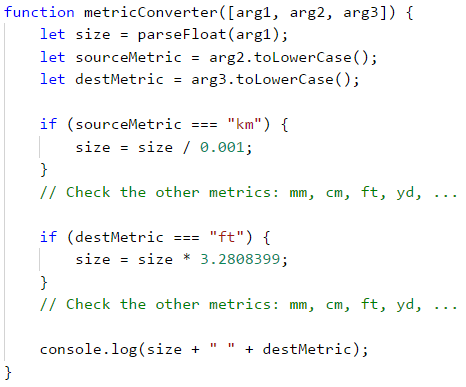
* First: A number.
* Second: Input unit of measure.
* Third: Output unit of measure (for the result).

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 12 km ft | 39370.0788 |
| 150 mi in | 9503999.99393599 |
| 450 yd km | 0.41147999937455 |

#### Hints and Guidelines

We take the input data and to the units of measure, we can add the method **toLowerCase()**, which will convert all letters to lower case. As we can see from the conversion table above, we have data for **converting only between meters and any other measuring unit**. To make the conversion, firstly we must calculate the input measurement in meters. To this effect, we need to create a set of conditionals to determine the input measuring unit and then the output.



#### Testing in The Judge System

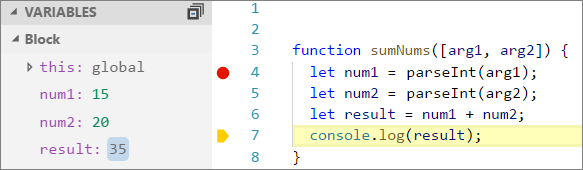
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#7>.

## Debugging - Simple Operations With a Debugger

To date, we have written quite a lot of code and oftentimes there were mistakes, were there? Now we can show you a tool to make finding mistakes easier.

### What is "Debugging"?

**Debugging** is the process of „**attaching**“ to a program's execution, which allows us to follow closely the execution of our program. We can follow **line by line** the events in our program, what is its evaluation route, what are the intermediate values of the declared variables at each step of the execution, among other useful information and thus allowing us to locate errors - the so-called **bugs**.



### Debugging in Visual Studio Code

We add a point, at which our function will stop its execution - a **breakpoint** after which we start the program in **debug mode** through pressing [**F5**]. The program will follow its course until it reaches our interrupt (**breakpoint**). After which we the execution can proceed to the **next line of code** by pressing [**F10**].

### Problem: Password Guess

Write a function that **accepts a password** (one line of random text) and checks if the input **matches** the phrase “**s3cr3t!P@ssw0rd**”. If it matches, print “**Welcome**”, otherwise print “**Wrong password!**”.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| qwerty | Wrong password! |
| s3cr3t!P@ssw0rd | Welcome |
| s3cr3t!p@ss | Wrong password! |

#### Hints and Guidelines

Use an **if-else** statement.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#8>.

### Problem: Number 100...200

Write a function that **accepts an integer** as a parameter and checks if it is **below 100**, **between 100 and 200**, or **over 200**. Print the appropriate messages as per the examples below.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 95 | Less than 100 |
| 120 | Between 100 and 200 |
| 210 | Greater than 200 |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#9>.

### Problem: Equal Words

Write a function that **accepts two words** as parameters and checks if they are the same. A comparison should be case-insensitive and the output should be either “**yes**” or “**no**”.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| Hello Hello | yes |
| SoftUni softuni | yes |
| Soft Uni | no |
| beer vodka | no |
| HeLlO hELLo | yes |

#### Hints and Guidelines

Before the comparison, both words should be in lower case, so that case (uppercase / lowercase) does not influence the result: **word = word.toLowerCase()**.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#10>.

### Problem: Speed Info

Write a function, that **takes speed** (decimal number) as a parameter and prints **speed information**. For speeds **up to 10** (inclusive), print "**slow**". For speed **over 10** and **up to 50**, print "**average**". For speeds **over 50** and **up to 150**, print "**fast**". For speeds **over 150** and **up to 1000**, print "**ultra fast**". For higher speed, print "**extremely fast**".

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 8 | slow |
| 49.5 | average |
| 126 | fast |
| 160 | ultra fast |
| 3500 | extremely fast |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#11>.

### Problem: Areas of Figures

Write a function that takes **the measures of a geometric shape** and **calculates its surface area**. There are four types of shapes: **square, rectangle. circle** and **triangle**.

The first argument of the function is the type of shape (**square**, **rectangle**, **circle**, **triangle**).

* If the shape is a **square**, the next argument will be one number - the length of its side.
* If the shape is a **rectangle**, the next argument will be two numbers - the lengths of its sides.
* If the shape is a **circle**, the next argument will be one number - the radius of the circle.
* If the shape is a **triangle**, the next argument will be two numbers - base and the corresponding altitude.

The result should be rounded up to the **third decimal point**.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| square 5 | 25 |
| rectangle 7 2.5 | 17.5 |
| circle 6 | 113.097 |
| triangle 4.5 20 | 45 |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#12>.

### Problem: Time + 15 Minutes

Write a function that takes **two parameters - hours and minutes** based on a 24-hour day and calculates what will be the time **after 15 minutes**. The result should be printed in the following format **hh:mm**. Hours should always be between 0 and 23, while minutes should always be between 0 and 59. Hours should be written with one or two digits as needed, while the minutes should always be written with two digits - add a **leading zero** as needed.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 1 46 | 2:01 |
| 0 01 | 0:16 |
| 23 59 | 0:14 |
| 11 08 | 11:23 |
| 12 49 | 13:04 |

#### Hints and Guidelines

Add 15 minutes and check using a set of conditions. If minutes are over 59 **increase the hours** by 1 and **decrease the minutes** by 60. You may handle the case when hours are over 23 similarly. Take care when printing the minutes to add a **leading zero** where appropriate.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#13>.

### Problem: 3 Equal Numbers

Write a function that takes **3 numbers** as arguments and prints whether they are the same (**yes** / **no**).

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 5 5 5 | yes |
| 5 4 5 | no |
| 1 2 3 | no |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#14>.

### Problem: Number 0...100 to Text

Write a function that converts numbers in the range of [**0 … 100**] in text.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 25 | twenty five |
| 42 | forty two |
| 6 | six |

#### Hints and Guidelines

Firstly you should check for **single-digit numbers** and if this is the case, print the corresponding word. Then you can check if the number is a **double-digit number**. These can be printed in two parts: left part (**tens** = number / 10) and right part (**units** = number % 10). If the number has three digits, then it must be 100 and this can be handled as a special case.

#### Testing in The Judge System

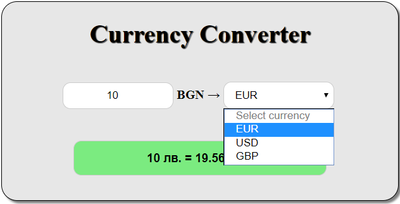
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/929#15>.

## Graphical Web Application

Now since we have completed a few exercises on **conditional statements (checks)**, let's do something a bit more interesting: an application with a Graphical User Interface (GUI) for currency conversion. We will employ the knowledge from this chapter to select from the different available currencies and make calculations as per the appropriate exchange rates for the given currency.

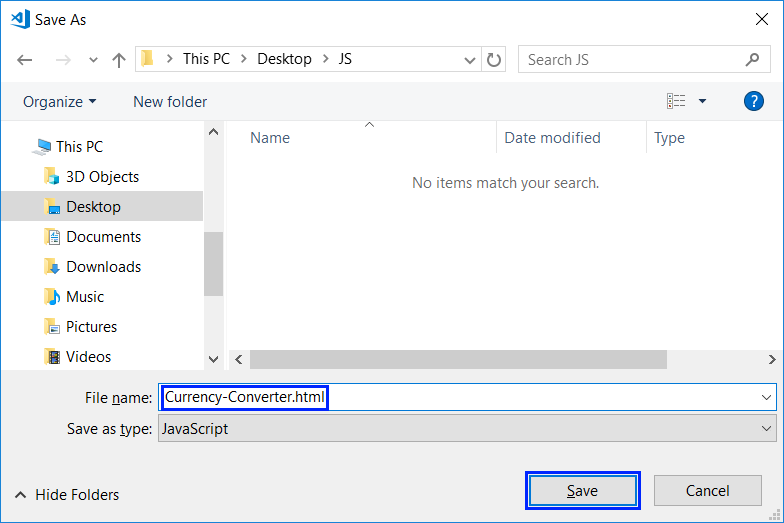
### Problem: \*\* Currency Converter

Now let's see how to create a graphical (**GUI**) application for **currency conversion**. The application will look similar to the picture below:



For visualization, we will use an **internet browser**, that interprets **HTML** pages. We will create a new page and will build the **structure**, **appearance**, and **functionality** of our application.

As usual, we **create a new file**, save it with the name **Currency-Converter**, however this time we add the file extension **.html**.



We open the newly created file and input the **document structure**, as **HTML code**:

<!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 styling here \*/

</style>

</head>

<body>

<main id="conventer-window">

<h1>Currency Converter</h1>

<form name="conventer">

<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>

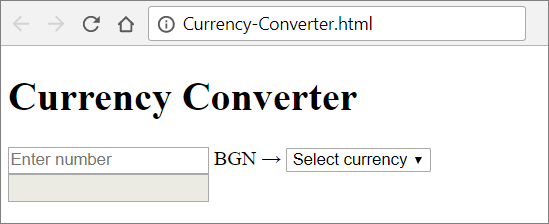
// enter JavaScript functionality here

</script>

</body>

</html>

We save the file and open it in the internet browser.



At this point we already have the document skeleton, but it can be visually enhanced through the addition of **styling**. To do this we add the following code in the **<style>** section for our **HTML** document:

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;

}

Now if we save and hit refresh, our application should look much better. All that is left now is to add the **functionality**. This is done through the **<script>** section of our **HTML** document. We will use the following **JavaScript code** to handle different user inputs and events:

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 + " " + "BGN = " + (x \* 1.95583).toFixed(2) + " " + selected;

document.getElementById("result").value = result;

} else if (selected === "USD") {

result = x + " " + "BGN = " + (x \* 1.63760).toFixed(2) + " " + selected;

document.getElementById("result").value = result;

} else if (selected === "GBP") {

result = x + " " + "BGN = " + (x \* 2.22920).toFixed(2) + " " + selected;

document.getElementById("result").value = result;

}

}

The above code takes the **amount** to be converted from the field **cash-input** and the **chosen currency** for the result of the field **currency-options**. After that with the use of a **conditional statement**, depending on the chosen currency, the amount is divided by the **exchange rate** (which is hard-coded by you in the source code). Finally, we generate a **text message with the result** (rounded to the second digit after the decimal point) and we output the message in the green box **result**. Have a go!

If you have problems with the examples above, **watch the video** at the beginning of this chapter or ask for help in the SoftUni official **Reddit**: <https://www.reddit.com/r/softuni/>.

# Chapter 3.2. Simple Conditions – Exam Problems

In the previous chapter, we went through the simple conditional statements in **JavaScript**, which we can use to execute different actions depending on a given condition. We mentioned what is the scope of a variable (it's **scope**) and how to track the execution of our program step by step (the so-called **debugging**). In this chapter, working with simple conditions by going through some exam tasks. To do this, let's first revise their construction:

if (bool expression) {

// condition body;

} else {

// else-construction body;

}

**if conditions** consist of:

* **if clause**
* **bool expression** - a variable of bool type (**Boolean**) or bool logical expression (an expression that results in **true/false**)
* **condition body** - contains a random block of source code
* **else clause** and its block of source code (**optional**)

## Exam Problems

After having revised how to write simple conditions, let's solve a few exam problems to practice the **if-else** construction:

## Problem: Transport Price

A student has to travel **n kilometers**. He can choose between **three types of transportation**:

* **Taxi**. Starting fee: **0.70** BGN. Day rate: **0.79** BGN/km. Night rate: **0.90** BGN/km.
* **Bus**. Day / Night rate: **0.09** BGN/km. Can be used for distances of a minimum of **20** km.
* **Train**. Day / Night rate: **0.06** BGN/km. Can be used for distances of a minimum of **100** km.

Write a program that reads the number of **kilometers n** and **period of the day** (day or night) and calculates **the price for the cheapest transport**.

### Input Data

**Two lines** (arguments) are read from the console:

* The first line (arguments) contains a number **n** – number of kilometers – an integer in the range of [**1 … 5000**].
* The second line contains the word "**day**" or "**night**" – traveling during the day or the night.

### Output Data

Print on the console **the lowest price** for the given number of kilometers.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 day | 4.65 | 7 night | 7 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 25 day | 2.25 | 180 night | 10.8 |

### Hints and Guidelines

We will read the input data and depending on the distance, we will choose the cheapest transport. To do that, we will write a few conditional statements.

#### Processing The Input Data

In the task, we are given **information about the input and output data**. Therefore, the first part of the task is to declare and initialize two **variables** where we will store the **values of the input data**:



Before starting with the conditional statements, we need to **declare** one more **variable** that stores the value of **the transport price**:



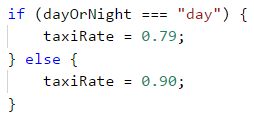
#### Checking The Conditions and Calculating

After having **declared and initialized** the input data and the variable that stores the value of the price, we have to decide which **conditions** of the task have to be **checked first**.

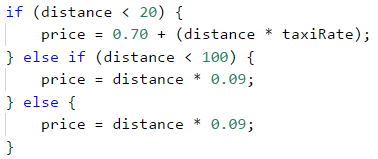
The task specifies that the rates of two of the vehicles **do not depend** on whether it is **day** or **night**, but the rate of one of the transports (taxi) **depends**. This is why the **first condition** will be whether it is **day or night** so that it is clear which rate the taxi will be **using**. To do that, we **declare one more variable** that stores **the value of the taxi rate**:



To calculate **the taxi rate**, we will use a conditional statement of type **if-else**:



After having done that, now we can start calculating **the transport price** itself. The constraints in the task refer to **the distance** that the student wants to travel. This is why, we will use an **if-else** statement that will help us find **the price** of the transport, depending on the given kilometers:



First, we check whether the kilometers are **less than 20**, as the task specifies that the student can only use **a taxi** for **less than 20 kilometers**. If the condition is **true** (returns **true**), the variable that is created to store the value of the transport (**price**), will **store** the corresponding value. This value equals **the starting fee** that we will **sum** with its **rate**, **multiplied** by **the distance** that the student has to travel.

If the condition of the variable **is not true** (returns **false**), the next step of our program is to check whether the kilometers are **less than 100**. We do that because the task specifies that in this range, **a bus** can be used as well. **The price** per kilometer of a bus **is cheaper** than a taxi one. Therefore, if the result of the condition is **true**, we store **a value**, equal to the result of the **multiplication** of **the rate** of the bus by **the distance** to the variable for the transportation **price** in the **else if** statement body.

If this condition **does not return true** as a result, we have to store **a value**, equal to **the result** of **the multiplication** of **the distance** by the train **rate** to the price variable in the **else** body. This is done because the train is **the cheapest** transport for the given distance.

#### Printing The Output

After we have checked the distance conditions and we have **calculated the price of the cheapest transport**, we have to **print it**. The task **does not** specify how to format the result, therefore, we just print **the variable**:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/930#0>.

## Problem: Pipes In Pool

A pool with **volume V** fills up via **two pipes**. **Each pipe has a certain flow rate** (the liters of water, flowing through a pipe for an hour). A worker starts the pipes simultaneously and goes out for **N hours**. Write a program that finds the state of the pool **the moment the worker comes back**.

### Input Data

**Four numbers are passed to the function** (arguments):

* The first line (argument) contains a number **V – the volume of the pool in liters** – an integer in the range of [**1 … 10000**].
* The second line (argument) contains a number **P1 – the flow rate of the first pipe per hour** – an integer in the range of [**1 … 5000**].
* The third line (argument) contains a number **P2 – the flow rate of the second pipe per hour** – an integer in the range of [**1 … 5000**].
* The fourth line (argument) contains a number **H – the hours that the worker is absent** – a floating-point number in the range of [**1.0 … 24.00**].

### Output Data

Print on the console **one of the two possible states**:

* To what extent the pool has filled up and how many percent each pipe has contributed. All percent values must be formatted to an integer (without rounding).
  + "The pool is **[x]**% full. Pipe 1: **[y]**%. Pipe 2: **[z]**%."
* If the pool has overflown – with how many liters it has overflown for the given time – a floating-point number.
  + "For **[x]** hours the pool overflows with **[y]** liters."

**Have in mind** that due to **the rounding to an integer**, there is **data loss** and it is normal for **the sum of the percents to be 99%, not 100%**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1000 100 120 3 | The pool is 66% full. Pipe 1: 45%. Pipe 2: 54%. | 100 100 100 2.5 | For 2.5 hours the pool overflows with 400 liters. |

### Hints and Guidelines

To solve the task, we read the input data, write a few conditional statements, do some calculations and print the result.

#### Processing The Input Data

Our first step is to read the input data:



Our next step is to **declare and initialize** a variable in which we are going to calculate how many **liters** the pool has **filled up** for the **time** the worker was **absent**. We do the calculations by **summing** the values of the flow rates of the **two pipes** and **multiplying** them by the **hours** that are given as input data:

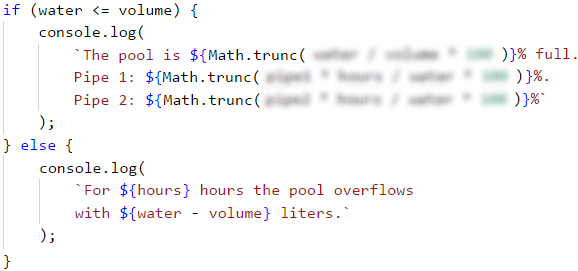


#### Checking The Conditions and Processing Output Data

After we have **the value of the quantity** of water that has flown through the **pipes**, the next step is to **compare** that quantity with the volume of the pool itself.

We do that with a simple **if-else** statement, where the condition will be whether **the quantity of water is less than the volume of the pool**. If the statement returns **true**, we have to print one **line** that contains **the ratio** between the quantity of **water that has flown through the pipes** and **the volume of the pool**, as well as the **ratio of the quantity of the water** from **each pipe** to the **volume of the pool**.

The ratio has to be in **percentage**, that is why all the calculations so far will be **multiplied by 100**. The values will be printed using **placeholders**, and as there is a condition for **the result in percentage** to be formatted to **two digits** after **the decimal** point **without rounding**, we will use the method **Math.trunc(…)**:



However, if **the condition** returns **false**, that means that **the quantity of water** is **more** than the **volume** of the pool, therefore, it has **overflown**. Again, the output data has to be on **one line**, but this time it should contain only two values – one of the **hours** when the worker was absent, and the **quantity of water**, which is the **difference** between \*the incoming water and the volume of the pool.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/930#1>.

## Problem: Sleepy Tom Cat

Tom Cat likes to sleep all day but, unfortunately, his owner is always playing with him whenever he has free time. To sleep well, **the norm of games** that Tom has is **30 000 minutes per year**. The time for games he has **depends on the holidays that his owner has**:

* During **workdays**, his owner plays with him **63 minutes per day**.
* During **holidays**, his owner plays with him **127 minutes per day**.

Write a program that reads **the number of holidays** and prints whether **Tom can sleep well** and how much **the difference from the norm** for the current year is. It is assumed that **there are 365 days in one year**.

**Example**: 20 holidays -> the working days are 345 (365 - 20 = 345). The time for games is 24 275 minutes (345 \* 63 + 20 \* 127). The difference from the norm is 5 725 minutes (30 000 – 24 275 = 5 725) or 95 hours and 25 minutes.

### Input Data

The input is read from the console and consists of an integer – **the number of holidays** in the range of [**0 … 365**].

### Output Data

**Two lines** have to be printed on the console:

* If Tom's time for games **is above the norm** for the current year:
  + **On the first line** print: **“Tom will run away”**.
  + **On the second line** print the difference from the norm in the format:  
    **“{H} hours and {M} minutes more for play”**.
* If the time for games of Tom **is below the norm** for the current year:
  + **On the first line** print: **“Tom sleeps well”**.
  + **On the second line** print the difference from the norm in the format:  
    **“{H} hours and {M} minutes less for play”**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 20 | Tom sleeps well 95 hours and 25 minutes less for play | 113 | Tom will run away 3 hours and 47 minutes more for play |

### Hints and Guidelines

To solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

#### Reading The Input Data and Calculating

From the task, we see that **the input data** will be **an integer** in the range of [**0 … 365**].



To solve the problem, **first**, we have to calculate **the total minutes** the owner of Tom is playing with him. We see that not only does the sleepy cat has to play with his owner during **the holidays**, but also during **the working days**. **The number** that we read from the console refers to **the holidays**.

Out next step is to **calculate**, with the help of that number, how many **the working days** of the owner are, as without them we cannot calculate **the total minutes for play**. As the total number of days per year is **365** and the number of holidays is **X**, that means that the number of working days is **365 - X\***. We store **the difference** in a new variable that only stores this value:

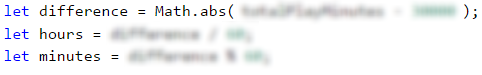


Once we have the number of days for playing, we can calculate **the time for games** of Tom in minutes. Its **value is equal** to the **result of the multiplication of the working days by 63** minutes (the task specifies that during working days, the time for play is 63 minutes per day), **summed with the result of the multiplication of the holidays by 127** minutes (the task specifies that during holidays, the time for play is 127 minutes per day).



In the task condition, we see that we have to **print the difference** between the two values in **hours** and **minutes** as output data. That is why we **subtract** the **total** time for play from the norm of **30 000** minutes and **store** the result in a **new** variable. After that, we **divide** that variable by 60 to get the **hours**, and then, to find out how many the **minutes** are, we use **modular division with the operator %**, as again we divide the variable of the difference by 60.

Here we have to note that if the total **time for the playing** of Tom is **less** than **30,000** when **subtracting** the norm from it, we will obtain **a negative number**. To **neutralize** the number in the division, we use **the method Math.abs(…)** when finding the difference:

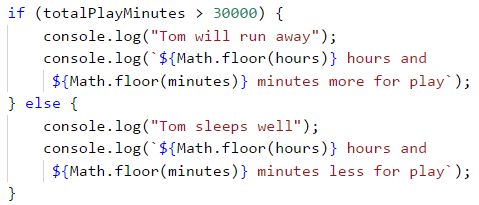


#### Checking The Conditions

The time for games is already calculated, which leads us to the **next** step – **comparing** the **time for play** of Tom with the **norm** on which the good sleep of the cat depends. To do so, we will use an **if-else** conditional statement. In the **if clause** we will check whether **the time for play is more than 30 000** (the norm).

#### Processing the Output Data

Whatever the result of the conditional statement is, we have to print how much **the difference in hours and minutes** is. We will do that with a **placeholder** and the variables that store the values of the hours and the minutes, as the formatting will be according to the task requirements for output.



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/930#2>.

## Problem: Harvest

In a vineyard with an area of X square meters, **40% of the harvest goes for wine production**. **Y kilograms of grapes** are extracted from a **1 square meters vineyard**. **2,5 kg of grapes** is needed for **1 liter of wine**. The wanted quantity of wine for sale is **Z liters**.

Write a program that **calculates how much wine can be produced** and whether that quantity is enough. **If it is enough, the rest is divided between the vineyard workers equally**.

### Input Data

The input data consists of **exactly 4 lines** (arguments):

* First line (argument): **X square meters is the vineyard size – an integer in the range of** [**10 … 5000**].
* Second line (argument): **Y grapes for one square meters – an integer in the range of** [**0.00 … 10.00**].
* Third line (argument): **Z needed liters of wine – an integer in the range of** [**10 … 600**].
* Fourth line (argument): **number of workers – an integer in the range of** [**1 … 20**].

### Output Data

The following has to be printed on the console:

* If the **produced** wine is **less than the needed quantity**:
  + **“It will be a tough winter! More {insufficient wine} liters wine needed.**”  
    \* **The result** has to be **rounded down to the nearest integer**.
* If **the produced** wine is **more than the needed quantity**:
  + **“Good harvest this year! Total wine: {total wine} liters.”**  
    \* **The result** has to be **rounded down to the nearest integer**.
  + **“{Wine left} liters left -> {wine for one worker} liters per person.”**  
    \* **Both of the results** have to be **rounded up to the higher integer**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 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. |

### Hints and Guidelines

To solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

#### Processing The Input Data and Performing The Calculations

First, we have to **read the input data**:



To solve the problem, based on the input data, we have to **calculate** how many **liters of wine** will be produced. From the task requirements, we see that to **calculate** the quantity of **wine in liters**, we first, have to find **the number of grapes in kilograms**, which we will get from the harvest. For that, we will declare a variable that keeps a **value**, equal to **40%** of the result from the **multiplication** of the vineyard area by the number of grapes, which is extracted from 1 �2*m*​2​​.

After having done these calculations, we are ready to **calculate the quantity of wine in liters** that will be produced from the harvest as well. For that, we declare one more variable that stores that **quantity**. To calculate, we have to **divide the number of grapes in kg by 2.5**:

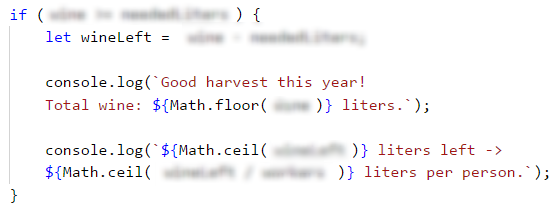


#### Checking The Conditions and Processing Output Data

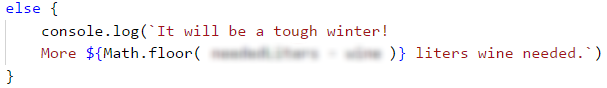
After having done the necessary calculations, the next step is to **check** whether the liters of wine that have been produced, **are enough**. For that, we will use **a simple conditional statement** of the **if-else** type and we will check whether the liters of wine from the harvest are **more than** or **equal to** the **needed liters**.

If the condition returns **true**, from the task requirement we see that **on the first line** we have to print **the wine that has been produced from the harvest**. That value has to be **rounded down to the nearest integer**, which we will do by using both the method **Math.floor(…)** and a **placeholder** when printing it.

On the second line, we have to print the results by **rounding them up to the higher integer**, which we will do by using the method **Math.ceil(…)**. The values that we have to print are **the quantity of wine left** and **the quantity that each worker gets**. The wine left is equal to **the difference** between the produced liters of wine and the needed liters of wine. We calculate the value of that quantity in a new variable, which we declare and initialize in the **if condition body**, before printing the first line. We calculate the quantity of wine that **each worker gets** by dividing the wine left by the number of workers.



If the condition returns **false**, we have to **print the difference** between **the needed liters** and the **liters of wine produced from the harvest**. There is a specification that the result has to be **rounded down to the nearest integer**, which we will do by using the method **Math.floor(…)**.



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/930#3>.

## Problem: Firm

A firm gets a request for creating a project for which a certain number of hours are needed. The firm has **a certain number of days**. During 10% of the days, the workers are **being trained** and cannot work on the project. A normal **working day is 8 hours long**. The project is important for the firm and every worker must work on it with **overtime of 2 hours per day**.

**The hours** must be **rounded down to the nearest integer** (for example, **6.98 hours** are rounded to **6 hours**).

Write a program that calculates whether **the firm can finish the project on time** and **how many hours more are needed or left**.

### Input Data

The input data contains **exactly three lines** (arguments):

* On **the first** line (argument) are **the needed hours** – **an integer in the range of** [**0 … 200 000**].
* On **the second** line (argument) are **the days that the firm has** – **an integer in the range of** [**0 … 20 000**].
* On **the third** line (argument) are **the number of all workers** – **an integer in the range of** [**0 … 200**].

### Output Data

Print **one line** on **the console**:

* If **the time is enough**:
  + **"Yes!{the hours left} hours left."**
* If **the time is NOT enough**:
  + **"Not enough time!{additional hours} hours needed."**

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 90 7 3 | Yes!99 hours left. | 99 3 1 | Not enough time!72 hours needed. |

### Hints and Guidelines

To solve the problem, we will read the input data. Then, we will write a few conditional statements and do some calculations. Finally, we will print the result.

#### Reading The Input Data

**First**, we have to read the input data to solve the problem.



#### Auxiliary Calculations

The next step is to calculate **the number of total working hours** by multiplying the working days by 8 (every working day is 8 hours long) with the number of workers and then sum them with the overtime. **The working days** equal **90% of the days** that the firm has. **The overtime** equals the result of the multiplication of the number of workers by 2 (the possible hours of overtime) and then it is multiplied by the number of days that the firm has. From the task requirements, we see that **the hours** should be **rounded down to the nearest integer**, which we will do with the method **Math.floor(…)**.

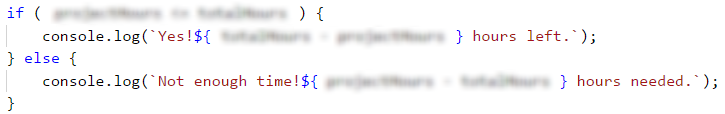


#### Checking The Conditions

After having done the calculations that are needed to find the value of **the working hours**, now we have to check whether these hours are **enough**, **or some hours are left**.

If **the time is enough**, we print the result that is specified in the task requirements, which in this case is the difference between **the working hours and the hours needed** for finishing the project.

If **the time is not enough**, we print the additional hours that are needed for finishing the project. They equal the difference between **the hours for the project** and **the total working hours**.



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/930#4>.

# Chapter 4.1. More Complex Conditions

In this **current** chapter, we are going to examine **nested conditional statements** in the **JavaScript** language, by which our program can execute **conditions**, that contain other **nested conditional statements**. We call them **"nested"** because **we put the if condition** into **another if condition**. We are going to examine the **more complex logical conditions** through proper examples.

## Nested Conditions

Pretty often the program logic requires the use of **if** or **if-else** statements, which are contained one inside another. They are called **nested** **if** or **if-else** statements. As implied by the title **"nested"**,these are **if** or **if-else** statements, that are placed inside other **if** or **else** statements.

if (condition1) {

if (condition2) {

// body;

} else {

// body;

}

}

Using more than three levels of nested conditional statements is not considered a good practice and has to be avoided, mostly through optimization of the structure/the algorithm of the code and/or by using another type of conditional statement, which we are going to examine below in this chapter.

### Problem: Personal Titles

Depending on **age** (decimal number) and **gender** (**m** / **f**) print a personal title:

* “**Mr.**” – man (gender “**m**”) 16 or more years old.
* “**Master**” – boy (gender “**m**”) under 16 years.
* “**Ms.**” – woman (gender “**f**”) 16 or more years old.
* “**Miss**” – girl (gender “**f**”) under 16 years.

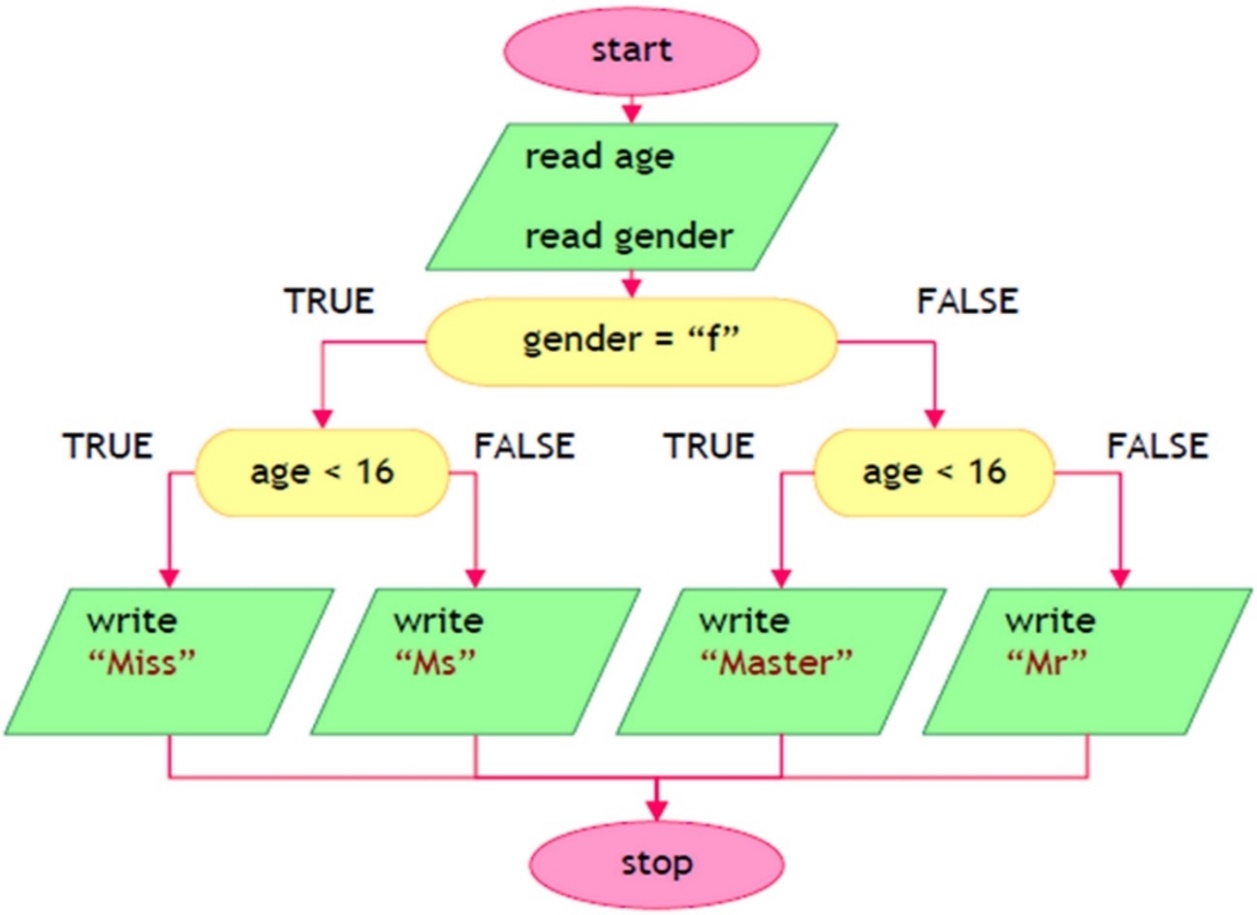
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 12 f | Miss | 17 m | Mr. |

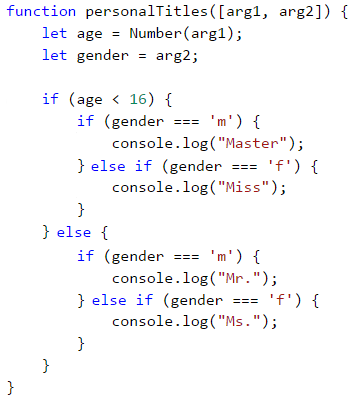
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 25 f | Ms. | 13.5 m | Master |

#### Solution

We should notice that the **output** of the program **depends on a few things**. First, we have to check what **gender** entered and then to check the **age**. Respectively, we are going to use **a few** **if-else** blocks. These blocks will be **nested**, i.e. from the **result** of the first, we are going to **define** which of the **others** to execute.



After reading the input data from the console, the following program logic should be executed:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#0>.

### Problem: Small Shop

A Bulgarian entrepreneur opens **small shops** in **a few** cities with different **prices** for the following **products**:

| **product / city** | **Sofia** | **Plovdiv** | **Varna** |
| --- | --- | --- | --- |
| coffee water beer sweets peanuts | 0.50 0.80 1.20 1.45 1.60 | 0.40 0.70 1.15 1.30 1.50 | 0.45 0.70 1.10 1.35 1.55 |

Calculate the price by the given **city** (string), **product** (string), and **quantity** (decimal number).

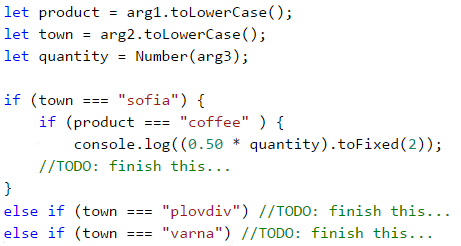
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| coffee Varna 2 | 0.90 | peanuts Plovdiv 1 | 1.50 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| beer Sofia 6 | 7.20 | water Plovdiv 3 | 2.10 |

#### Solution

We **convert** all of the letters into the **lower register** by using the method **.toLowerCase()**, to compare products and cities **no matter** what the letters are - small/capital ones.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#1>.

## More Complex Conditions

Let's take a look at how we can create more complex conditions. We can use the logical "**AND**" (**&&**), logical "**OR**" (**||**), logical **negation** (**!**) or **brackets** (**()**).

### Logical "AND"

As we saw, in some tasks we have to make **multiple checks at once**. But what happens, when executing some code, **more** conditions have to be executed and we **don't want to** make a **negation** (**else**) for each one of them? The option with the nested **if blocks** are valid, but the code will look very **unordered** and for sure - **hard** to read and maintain.

Logical "**AND**" (operator **&&**) means a few conditions have to be **fulfilled simultaneously**. The following table of truthfulness is applicable:

| **a** | **b** | **a && b** |
| --- | --- | --- |
| true true false false | true false true false | true false false false |

### How Does The && Operator Work?

The operator **&&** accepts **a couple of Boolean** (conditional) statements, which have a **true** or **false** value, and returns **one** boolean statement as a **result**. Using it **instead** of a couple of nested **if** blocks, makes the code **more readable**, **ordered**, and **easy** to maintain. But how does it **work**, when we put a **few** conditions one after another? As we saw above, the logical **"AND"** returns **true**, **only** when it accepts as **arguments statements** with the value **true**. Respectively, when we have a **sequence** of arguments, the logical "**AND**" checks either until one of the arguments is **over**, or until it **meets** an argument with value **false**.

**Example**:

let a = true;

let b = true;

let c = false;

let d = true;

let result = a && b && c && d;

// false (as d is not being checked)

The program will run in the **following** way: **It starts** the check from **a**, **reads** it, and accepts that it has a **true** value, after that, it **checks b**. After it has **accepted**, that **a** and **b** return value **true**, **it checks the next** argument. It gets to **c** and sees that the variable has a **false** value. After the program accepts that argument **c** has a **false** value, it calculates the expression **to c**, **regardless** of what the value of **d** is. That is why the evaluation of **d**is being **skipped** and the whole expression is calculated as **false**.

### Problem: Point in Rectangle

Checks whether **point {x, y}** is placed **inside the rectangle {x1, y1} – {x2, y2}**. The input data is readfrom the console and consists of 6 lines: the decimal numbers **x1**, **y1**, **x2**, **y2**, **x**, and **y** (as it is guaranteed that **x1 < x2** and **y1 < y2**).

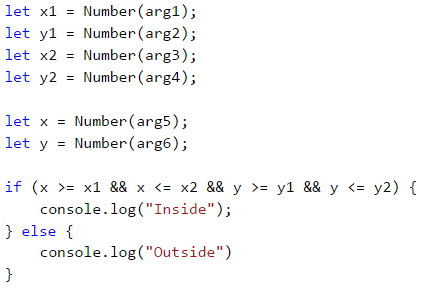
#### Sample Input and Output

| **Input** | **Output** | **Visualization** |
| --- | --- | --- |
| 2 -3 12 3 8 -1 | Inside | shop |

#### Solution

A point is internal for a given polygon if the following four conditions are applied at the same time:

* The point is placed to the right from the left side of the rectangle.
* The point is placed to the left from the right side of the rectangle.
* The point is placed downwards from the upper side of the rectangle.
* The point is placed upwards from the downside of the rectangle.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#2>.

## Logical "OR"

The logical **"OR"** (operator **||**) means that **at least one** among a few conditions is fulfilled. Similar to the operator **&&**, the logical **"OR"** accepts a few arguments of **boolean** (conditional) type and returns **true** or **false**. We can easily guess that we **obtain** as value **true**, whenever at least **one** of the arguments has a **true** value. A typical example of the logic of this operator is the following:

At school, the teacher says: "Ivan or Peter should clean the board". To fulfill this condition (to clean the board), either Ivan can clean it, or just for Peter to clean it, or both of them to clean it together.

| **a** | **b** | **a || b** |
| --- | --- | --- |
| true true false false | true false true false | true true true false |

### How Does The || Operator Work?

We have already learned what the logical **"OR"** **represents**. But how is it being achieved? Just like the logical **"AND"**, the program **checks** from left to right **the arguments**, that are given. To obtain **true** from the expression, just one argument must have a **true** value, respectively the checking **continues** until an **argument** with **such** value is met or until the arguments **are over**.

Here is one **example** of the operator **||** in action:

let a = false;

let b = true;

let c = false;

let d = true;

let result = a || b || c || d;

// true (as c and d are not being checked)

The program **checks a**, accepts that it has a value **false**, and continues. Reaching **b**, it understands that it has **true** value and the whole **expression** is calculated as **true**, **without** having to check **c** or **d**, because their values **wouldn't change** the result of the expression.

### Problem: Fruit or Vegetable

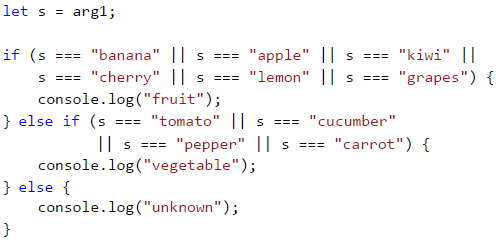
Let's check whether a given product is a **fruit** or a **vegetable**. The "**fruits**" are **banana**, **apple**, **kiwi**, **cherry**, **lemon**, and **grapes**. The "**vegetables**" are **tomato**, **cucumber**, **pepper**, and **carrot**. Everything else is "**unknown**".

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| banana tomato Java | fruit vegetable unknown |

#### Solution

We have to use a few conditional statements with logical "**OR**" (**||**):



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#3>.

## Logical Negation

**Logical negation** (operator **!**) means a given condition is **not fulfilled**.

| **a** | **!a** |
| --- | --- |
| true | false |

The operator **!** accepts as an **argument** a boolean variable and **returns** its value.

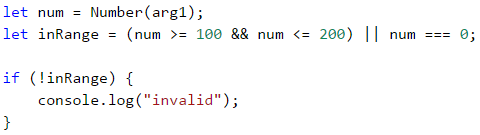
### Problem: Invalid Number

A given **number is valid** if it is in the range [**100 … 200**] or it is **0**. Validate an **invalid** number.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 75 | invalid |
| 150 | (no output) |
| 220 | invalid |

#### Solution



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#4>.

## The Parentheses () Operator

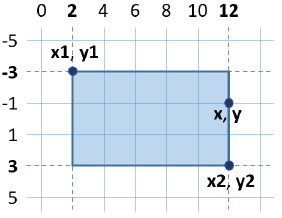
Like the other operators in programming, the operators **&&** and **||** have a priority, and in this case, the operator **&&** have higher precedence than the operator **||**. The operator **()** serves to **change the priority of the operator** and is calculated first, just like in math. The use of parentheses also gives the code better readability and is considered a good practice.

## More Complex Logical Conditions

Sometimes the conditions may be very complex, so they can require a long boolean expression or a sequence of conditions. Let's take a look at a few problems.

### Problem: Point on Rectangle Border

Write a program that checks whether a **point {x, y}** is placed **onto any of the sides of a rectangle {x1, y1} - {x2, y2}**. The input data is read from the console and consists of 6 lines: the decimal numbers **x1**, **y1**, **x2**, **y2**, **x**, and **y** (as it is guaranteed that **x1 < x2** and **y1 < y2**). Print "**Border**" (if the point lies on any of the sides) or "**Inside / Outside**" (in the opposite case).



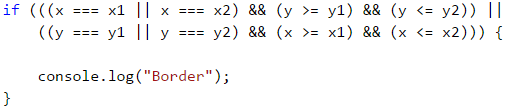
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 -3 12 3 12 -1 | Border | 2 -3 12 3 8 -1 | Inside / Outside |

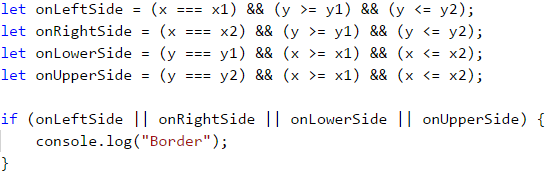
#### Solution

The point lies on any of the sides of the rectangle, if:

* **x** coincides with **x1** or **x2** and at the same time **y** is between **y1** and **y2** or
* **y** coincides with **y1** or **y2** and at the same time **x** is between **x1** and **x2**.



The previous evaluation might be simplified in the following way:



The second way with the additional boolean variables is longer, but is much more understandable than the first one, isn't it? We recommend when you write boolean conditions to make them **easy to read and understand**, instead of making them short. Use additional variables with meaningful names, if needed. The names of the boolean variables have to hint at what the value that is kept inside them represents.

What remains is to finish writing the code to print “**Inside / Outside**”, if the point is not onto any of the sides of the rectangle.

#### Testing in The Judge System

After you finish writing the solution, you can test it here: <https://judge.softuni.org/Contests/Practice/Index/931#5>.

### Problem: Fruit Shop

A fruit shop during **weekdays** sells at the following **prices**:

| **Fruit** | **Price** |
| --- | --- |
| banana apple orange grapefruit kiwi pineapple grapes | 2.50 1.20 0.85 1.45 2.70 5.50 3.85 |

During the **weekend days** the prices are higher:

| **Fruit** | **Price** |
| --- | --- |
| banana apple orange grapefruit kiwi pineapple grapes | 2.70 1.25 0.90 1.60 3.00 5.60 4.20 |

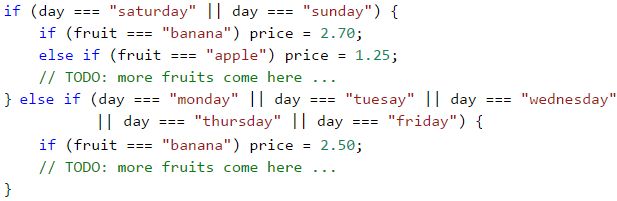
Write a program that reads from the console **fruit** (banana / apple / …), **a day of the week** (Monday / Tuesday / …), and **quantity (decimal number)** and **calculates the price** according to the prices from the tables above. The result has to be printed **rounded up to 2 digits after the decimal point**. Print **“error”** if it is an **invalid day** of the week or an **invalid name** of a fruit.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| orange Sunday 3 | 2.70 | kiwi Monday 2.5 | 6.75 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| grapes Saturday 0.5 | 2.10 | tomato Monday 0.5 | error |

#### Solution



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#6>.

### Problem: Trade Commissions

A company is giving the following **commissions** to its traders according to the **city**, in which they are working and the **volume of sales s**:

| **City** | **0 <= s <= 500** | **500 < s <= 1000** | **1000 < s <= 10000** | **s > 10000** |
| --- | --- | --- | --- | --- |
| Sofia Varna Plovdiv | 5% 4.5% 5.5% | 7% 7.5% 8% | 8% 10% 12% | 12% 13% 14.5% |

Write a program that reads the name of a **city** (string) and the volume of **sales** (decimal number) and calculates the rate of the commission fee. The result has to be shown rounded **up to 2 digits after the decimal point**. When there is an **invalid city or volume of sales** (a negative number), print "**error**".

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| Sofia 1500 | 120.00 | Plovdiv 499.99 | 27.50 | Kaspichan -50 | error |

#### Solution

When reading the input, we could convert the city into small letters (with the method **.toLowerCase()**). Initially, we set the commission fee to **-1**. It will be changed if the city and the price range are found in the table of commissions. To calculate the commission according to the city and volume of sales, we need a few nested **if statements**, as in the sample code below:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#7>.

|  |  |
| --- | --- |
|  | **It is a good practice** to use **blocks**, that are **enclosed** with curly braces **{ }** after **if** and **else**. Also, it is recommended during writing to **move aside** the code **after if and else** with a single tabulation **inward**, to make the code more easily readable. |

## Switch-Case Conditional Statements

The **switch-case** construction works as a sequence of **if-else** blocks. Whenever the work of our program depends on the value of **one variable**, instead of making consecutive with **if-else** blocks, we can **use** the conditional **switch-case** statement. It is being used for **choosing between a list of possibilities**. The statement compares a given value with defined constants and depending on the result, it takes an action.

We put **the variable**, that we want to **compare**, inside the **brackets after the operator switch** and it is called a "**selector**". Here **the type must be comparable** (numbers, strings). **Consecutively**, the program starts **comparing** each **value**, that **is found** after the **case labels**. Upon a match, the executions of the code from the respective place continue, until it reaches the **break** operator. When **no matches** are **found**, the **default** construction is being executed, **if** such **exist**.

switch (selector) {

case value1:

construction;

break;

case value2:

construction;

break;

case value3:

construction;

break;

…

default:

construction;

break;

}

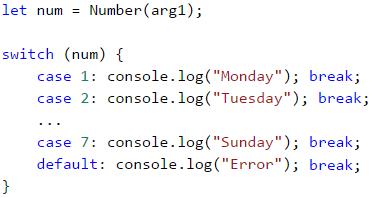
### Problem: Day of Week

Let's write a program that prints the **day of the week** (in English) depending on the **given number** (1 … 7) or "**Error**" if an invalid day is given.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 1 7 -1 | Monday Sunday Error |

#### Solution



|  |  |
| --- | --- |
|  | It is a good practice to put at **first** place those **case statements**, that process **the most common situations**, and leave the **case construction**, processing **the more rare situations**, in **the end, before the default constructions**. Another good practice is to **arrange the case labels** in **ascending order**, regardless of whether they are integral or symbolic. |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#8>.

### Multiple Labels in Switch-Cases

In **JavaScript**, we can **use multiple case** labels, when they have to execute **the same** code. In this way, when our program finds a **match**, it will execute **the next** code, because **after** the respective **case** label **there is no code** for execution and a **break** operator:

switch (selector) {

case value1:

case value2:

case value3:

construction;

break;

case value4:

case value5:

construction;

break;

…

default:

construction;

break;

}

### Problem: Animal Type

Write a program that prints the type of the animal depending on its name:

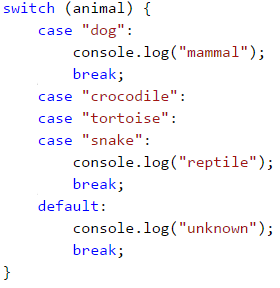
* dog -> **mammal**
* crocodile, tortoise, snake -> **reptile**
* others -> **unknown**

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| tortoise | reptile | dog | mammal | elephant | unknown |

#### Solution

We can solve the task with **switch-case** conditions with multiple labels in the following way:



#### Testing in The Judge System

Thes your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#9>.

## What Have We Learned from This Chapter?

Let's look back on what new constructions and programming techniques we have learned in this chapter:

### Nested Conditions

if (condition1) {

if (condition2) {

// body;

} else {

// body;

}

}

### More Complex Conditions with &&, ||, !, and ()

if ((x === left || x === right) && y >= top && y <= bottom)

console.log(…);

#### Switch-Case Statements

switch (selector) {

case value1:

construction;

break;

case value2:

case value3:

construction;

break;

…

default:

construction;

break;

}

## Exercises: More Complex Conditions

Now let's exercise our new skills with complex conditions. Let's solve a few practical tasks.

### Problem: Cinema

In a cinema hall, the chairs are ordered in a **rectangle** shape in **r** rows and **c** columns. There are three types of screening with tickets of **different** prices:

* **Premiere** – a premiere screening, with a price of **12.00** BGN.
* **Normal** – a standart screeneing, with a price of **7.50** BGN.
* **Discount** – a screening for children and students at a promotional price of **5.00** BGN.

Write a program that enters a **type of screening** (string), several **rows**, and several **columns** in the hall (integer numbers) and calculates **the total income** from tickets from a **full hall**. The result has to be printed in the same format as in the example below - rounded up to 2 digits after the decimal point.

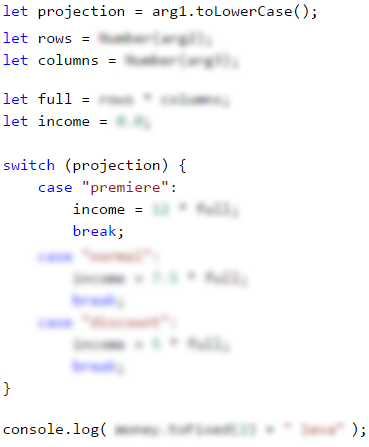
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| Premiere 10 12 | 1440.00 BGN | Normal 21 13 | 2047.50 BGN |

#### Hints and Guidelines

While reading the input, we could convert the screening type into small letters (with the method **.toLowerCase()**). We create and initialize a variable, that will store the calculated income. In another variable, we calculate the full capacity of the hall. We use a **switch-case** conditional statement, to calculate the income according to the type of projection and print the result on the console in the given format (look for the needed **JavaScript** functionality on the internet).

Sample code (parts of the code are blurred with the purpose to stimulate your thinking and solving skills):



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#10>.

### Problem: Volleyball

Vlady is a student, lives in Sofia, and goes to his hometown from time to time. He is very keen on volleyball, but he is very busy during weekdays and plays **volleyball** only during **weekends** and on **holidays**. Vlady plays **in Sofia** every **Saturday** when **he is not working**, and **he is not traveling to his hometown**, and also during **2/3 of the holidays**. He travels to his **hometown h times** a year, where he plays volleyball with his old friends on **Sunday**. Vlady **is not working 3/4 of the weekends**, during which he is in Sofia. Furthermore, during **leap years** Vlady plays **15% more** volleyball than usual. We accept that the year has exactly **48 weekends**, suitable for volleyball. Write a program, that calculates **how many times Vlady has played volleyball** through the year. **Round the result** down to the nearest integer (e.g. 2.15 -> 2; 9.95 -> 9).

The input data is read from the console:

* The first line contains the word “**leap**” (leap year) or “**normal**” (a year has 365 days).
* The second line contains the integer **p** – the count of holidays in the year (which are not Saturday or Sunday).
* The third line contains the integer **h** – the count of weekends, in which Vlady travels to his hometown.

#### Sample Input and Output

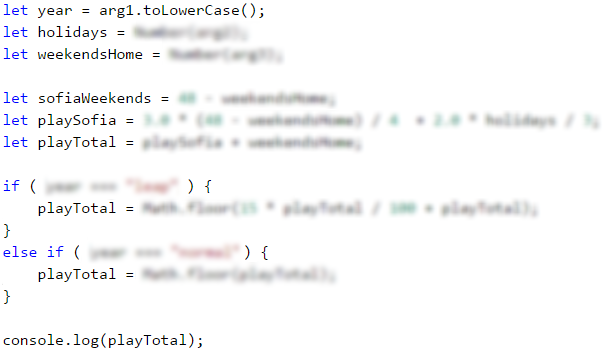
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| leap 5 2 | 45 | normal 3 2 | 38 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| normal 11 6 | 44 | leap 0 1 | 41 |

#### Hints and Guidelines

As usual, we read the input data from the console and, to avoid making mistakes, we convert the text into small letters with the method **.toLowerCase()**. Consequently, we calculate **the weekend spent in Sofia**, **The time for playing in Sofia**, and **the total playtime**. At last, we check whether the year is a **leap**, we make additional calculations when necessary and we print the result on the console, **rounded down** to the nearest **integer** (look for a **JavaScript** class with such functionality).

A sample code (parts of the code are blurred on purpose to stimulate independent thinking and solving skills):

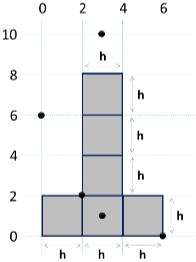


#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#11>.

### Problem: \* Point in Figure

The figure consists of **6 blocks with size h \* h**, placed as in the figure below. The lower left angle of the building is on position {0, 0}. The upper right angle of the figure is in position {**2\*h**, **4\*h**}. The coordinates given in the figure are for **h = 2**:



Write a program, that enters an integer **h** and the coordinates of given **point {x, y}** (integers) and prints whether the points are inside the figure (**inside**), outside the figure (**outside**), or on any of the borders of the figure (**border**).

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 3 10 | outside | 2 3 1 | inside |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 2 2 | border | 2 6 0 | border |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 0 6 | outside | 15 13 55 | outside |

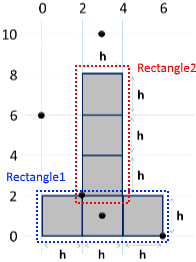
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 15 29 37 | inside | 15 37 18 | outside |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 15 -4 7 | outside | 15 30 0 | border |

#### Hints and Guidelines

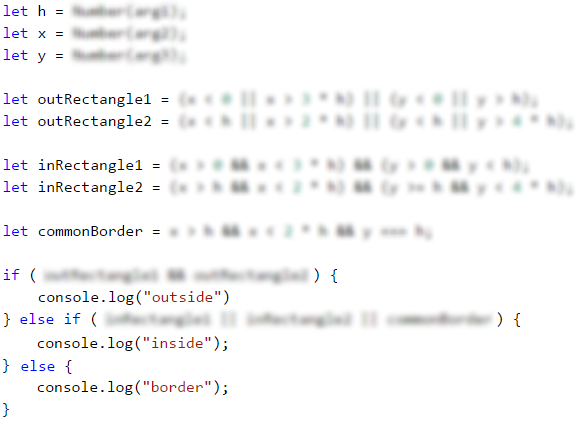
A possible logic for solving the task (not the only correct one):

* We might split the figure into **two rectangles** with a shared side:



* A point is **outer (outside)** for the figure when it is **outside** both of the rectangles.
* A point is **inner (inside)** for the figure if it is inside one of the rectangles (excluding their borders) or lies on their shared side.
* In **another case**, the point lies on the border of the rectangle (**border**).

Sample code (parts of the code are blurred to stimulate logical thinking and solving skills):



#### Testing in The Judge System

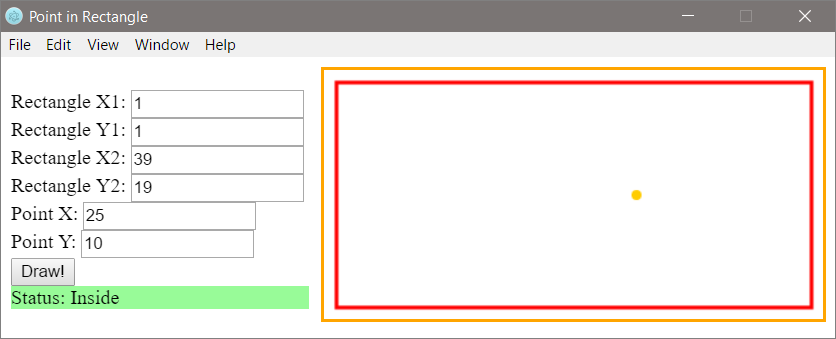
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/931#12>.

## Exercise: Graphic Application with More Complex Conditions

In this chapter, we learned how we can make **statements with non-trivial conditions**. Now let's apply this knowledge to create something interesting: a desktop application, that visualizes a point in a rectangle. This is a wonderful visualization of one of the tasks from the exercises.

### Lab: \* Point in a Rectangle – (GUI) Application

The task we have is to develop a graphical (**GUI**) application for **visualizing a point in a rectangle**. The application must look like identically the following:



Using the controls on the left we set the coordinates of **two of the angles of the rectangle** (decimal numbers) and the coordinates of the **point**. The application **visualizes graphically** the rectangle and the point and prints whether the point is **inside** the rectangle (**Inside**), **outside** of it (**Outside**), or on one of the sides (**Border**). The application **moves and resizes** the coordinates of the rectangle and the point to be maximum large, but to fit the field for visualization, on the right side of the application.

|  |  |
| --- | --- |
|  | **Attention**: this application is significantly **more complex** than the previous graphical applications, which we have to develop until now because it requires using functions for drawing (Canvas ), work with HTML, JavaScript, and GUI framework (Electron). |

Follow the instructions for building the application step by step:

1. We will first create a separate folder for the project of our application with a suitable name, for example: "**Point-in-Rectangle**".
2. **Electron** – working frame (**framework**) for creating graphical (**GUI**) applications with JavaScript. We execute the following command on the console (Command Prompt / Bash):
3. npm install -g electron
4. In the project folder, we create a **JavaScript file** named **main.js** using the keyboard shortcut [**Ctrl + N**] in VS Code. After that, we save the new file by using [**Ctrl + Shift + S**] and entering the desired file name.
5. The code, described in **main.js**, manages the events and creates new windows in the application. It should look like the following code below:

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', () => {

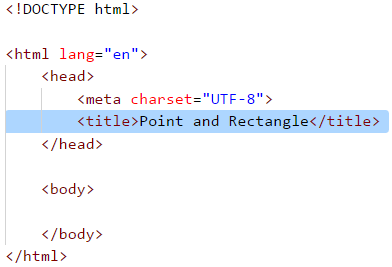
if (win === null) {

createWindow();

}

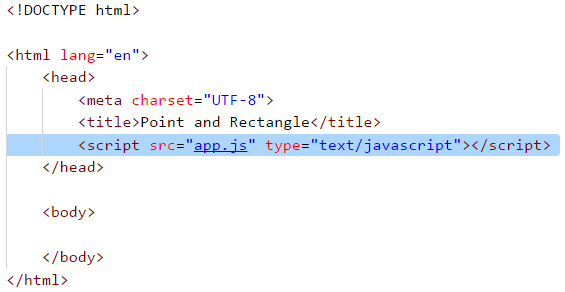
});

1. In the project folder, we create an **HTML** file with the name **index.html**. The **<title>** tag is mandatory for every **HTML** document and defines its title. We open it and write **"Point in Rectangle"**:



We add the following code under the **<title>** in the **index.html** file:

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



In this way the connection bettween the files **index.html** and **app.js** (which we will create later) is realized. The **<body>** tag defines **the body of the html document**. We write in it the following code:

<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">

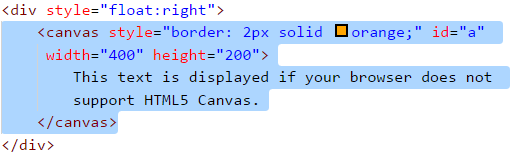
This text is displayed if your browser does not support HTML5 Canvas.

</canvas>

</div>

</body>

To enter the coordinates of the rectangle and the point, we use **input** fields of type **Number**, with tags **<label>**. To draw geometric figures in the application, we use the html tag **<canvas>**:



It accepts the following parameters:

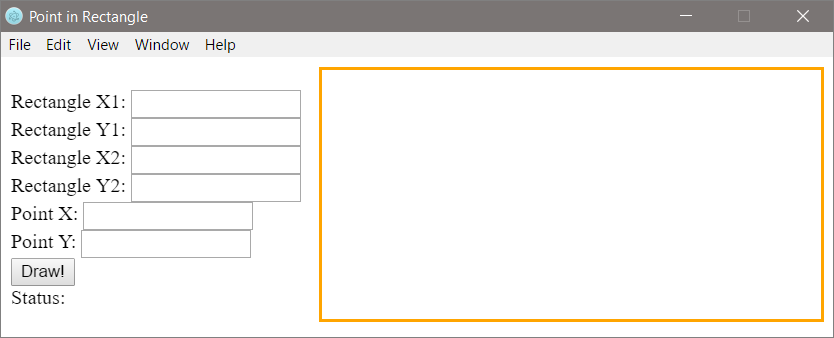
* **Width** (width) in pixels (px)
* **Height** (height) in pixels (px)
* **Border** (border)

To have your changes stored in the application, the files must be saved with **[Ctrl+S]**.

To start the application, run the following command in the console (in the folder of the current project):

electron .

The application should look like the following:



1. The most complicated part remains to be implemented: **visualization of the rectangle and the point** at the field of the element **<canvas>** by using the function **draw()** in the file **app.js**, which we create in the directory of the application, in the way described at Point 2.

We create [**CanvasRenderingContext2D**](https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D) an object by writing the following code:

// Create canvas element

let canvas = document.getElementById('a');

let context = canvas.getContext('2d');

The **<canvas>** element is a field, in which the object generated with the method [**.getContext('2d')**](https://developer.mozilla.org/en-US/docs/Web/API/HTMLCanvasElement/getContext), is drawing graphics, text, images and other elements. In this case, the **context** variable represents this object. We store in separate variables the coordinates of the two angles of the rectangle:

// 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;

The coordinate values are accesible through the **id** of the **<input>** fields. For better visualization of the screen, we scale the values by **increasing them 10 times**. The next step is to calculate the sides of the rectangle, because the object **context** draws a rectangle on four parameters: **x** - coordinate, **y** coordinate, **width** in pixels and **height** in pixels:

// Calculate rectangle parameters

let rectWidth = Math.abs(rectX1 - rectX2);

let rectHeight = Math.abs(rectY1 - rectY2);

We can use the code below, which draws a red rectangle, according to the coordinates given in the form, using the method [**.strokeRect(...)**](https://www.w3schools.com/tags/canvas_strokerect.asp):

// Set rectangle style

context.strokeStyle = "#ff0000";

context.lineWidth = 3;

// Draw rectangle with given parameters

context.strokeRect(rectX1, rectY1, rectWidth, rectHeight);

Similar to the rectangle, we take the coordinates of the point and scale them. After that, we set the style of the point - orange color. For better visualization of the screen, we convert the point into a circle with the method [**.arc(...)**](https://www.w3schools.com/tags/canvas_arc.asp). This method accepts five parameters: **x** - coordinate, **y** - coordinate, **radius**, **start of the arc** in radians, **end of the arc** radians:

// 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();

To reflect the results in the **if** checks, we store the following elements of the html code in separate variables:

// Assign variables to (<div id="result">) and (<span id="status">) html elements

let result = document.getElementById("status");

let output = document.getElementById("result");

The last step is to check the position of the point relative to the rectangle:

// 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";

}

Let's now think about how **to add** the unfinished (on purpose) conditions in the **if** constructions. The code above is intentionally not compiled, because its purpose is to make the reader think about how and why it works and to complete the missing parts. The code above takes the coordinates of the rectangle and the point checks whether the point is inside, outside, or on the side of the rectangle. When the result is visualized, the background color of the text block that contains it also changes.

This is the full version of the function **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";

}

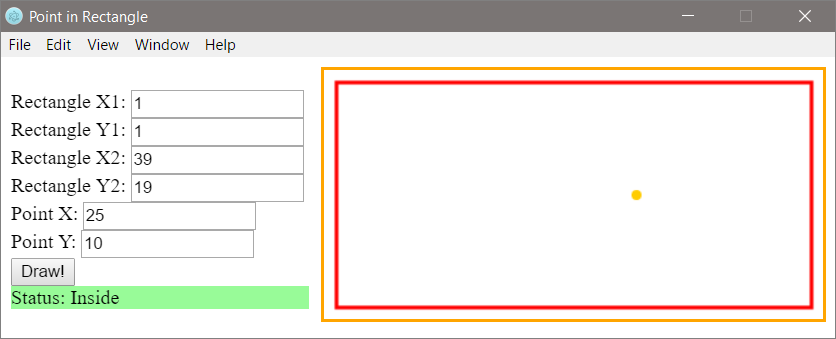
}

We start the application through the file **index.html** and test it (by entering the different input data). We try to enter different rectangles and locate the point at different positions, resize the application and see if it behaves correctly. If the application does not work correctly, we check for errors. The most likely cause for an error is if we wrote the code in the wrong place.

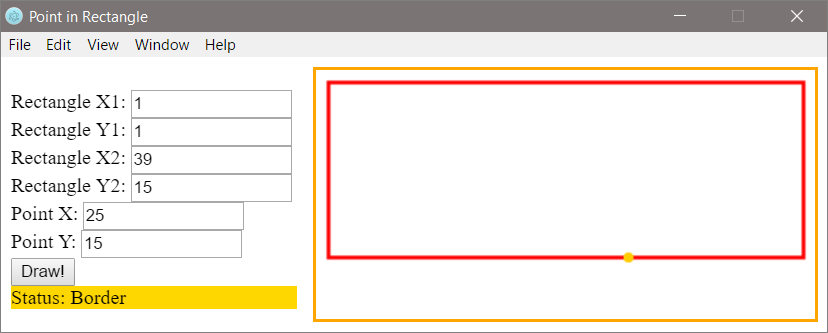
At last, we start the application in our own GUI window:

electron .

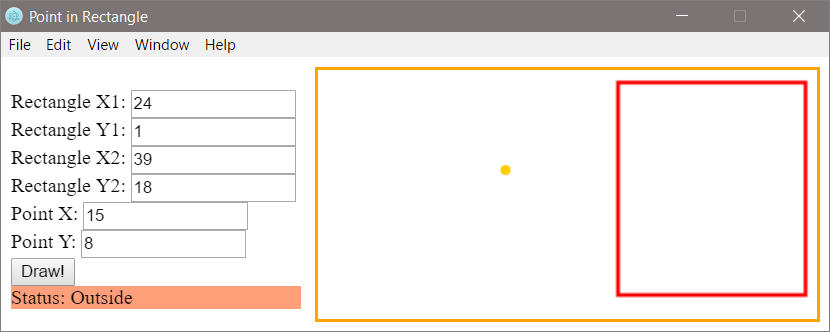
**Case 1: The Point is in The Rectangle**:



**Case 2: The Point Lies on One of The Sides of The Rectangle**:



**Case 3: The Point is Outside The Rectangle**:



If you have problems with the last task, feel free to ask in **the SoftUni Reddit**: <https://www.reddit.com/r/softuni/>.

# Chapter 4.2. More Complex Conditions – Exam Problems

The previous chapter introduced you to **nested conditions** in **JavaScript**. Via nested conditions, the program logic in a particular application can be represented using **if conditional statements** that are nested one into another. We also explained the **switch-case** conditional statement that allows selecting from a list of options. Now we are going to solve some practical exercises and make sure we have an in-depth understanding of the material, by discussing a set of more complex problems that had been given to students on exams. Before moving to the problems, let's first recall what nested conditions are:

## Nested Conditions

if (condition1) {

if (condition2)

// body;

else

// body;

}

|  |  |
| --- | --- |
|  | Remember that it is not a good practice to write **deeply nested conditional statements** (with more than three levels of nesting). Avoid nesting of more than three conditional statements inside one another. This complicates the code and makes its reading and understanding difficult. |

## Switch-Case Conditions

When the program operation depends on the value of a variable, instead of doing consecutive checks with multiple **if-else** blocks, we can use the **switch-case** conditional statement.

switch (selector) {

case value1:

statement;

break;

case value2:

statement;

break;

default:

statement;

break;

}

The structure consists of:

* Selector - an expression that calculates a particular value.
* Multiple **case** labels followed by commands, ending in a **break**.

## Exam Problems

Now, after we refreshed our knowledge on how to use and nested conditional statements to implement more complex conditions and program logic, let's solve some exam problems.

## Problem: On Time for The Exam

A student has to attend **an exam at a particular time** (for example at 9:30 am). They arrive in the exam room at a particular **time of arrival** (for example 9:40 am). It is considered that the student has arrived **on time** if they have arrived **at the time when the exam starts or up to half an hour earlier**. If the student has arrived **more than 30 minutes earlier**, the student has come **too early**. If they have arrived **after the time when the exam starts**, they are **late**.

Write a program that inputs the exam starting time and the time of student's arrival, and prints if the student has arrived **on time**, if they have arrived **early** or if they are **late**, as well as **how many hours or minutes** the student is early or late.

### Input Data

Read the following **four integers** (arguments):

* The first line (argument) contains the **exam starting time (hours)** – an integer from 0 to 23.
* The second line (argument) contains the **exam starting time (minutes)** – an integer from 0 to 59.
* The third line (argument) contains the **hour of arrival** – an integer from 0 to 23.
* The fourth line (argument) contains **minutes of arrival** – an integer from 0 to 59.

### Output Data

Print the following on the first line on the console:

* "**Late**", if the student arrives **later** compared to the exam starting time.
* "**On time**", if the student arrives **exactly** at the exam starting time or up to 30 minutes earlier.
* "**Early**", if the student arrives more than 30 minutes **before** the exam starting time.

If the student arrives with more than one minute difference compared to the exam starting time, print on the next line:

* "**mm minutes before the start**" for arriving less than an hour earlier.
* "**hh:mm hours before the start**" for arriving 1 hour or earlier. Always print minutes using 2 digits, for example "1:05".
* "**mm minutes after the start**" for arriving more than an hour late.
* "**hh:mm hours after the start**" for arriving late with 1 hour or more. Always print minutes using 2 digits, for example "1:03".

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 9 30 9 50 | Late 20 minutes after the start | 16 00 15 00 | Early 1:00 hours before the start |
| 9 00 8 30 | On time 30 minutes before the start | 9 00 10 30 | Late 1:30 hours after the start |
| 14 00 13 55 | On time 5 minutes before the start | 11 30 8 12 | Early 3:18 hours before the start |

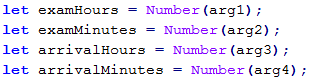
| **Input** | **Output** |
| --- | --- |
| 10 00 10 00 | On time |
| 11 30 10 55 | Early 35 minutes before the start |
| 11 30 12 29 | Late 59 minutes after the start |

### Hints and Guidelines

|  |  |
| --- | --- |
|  | It is recommended **to read the assignment a few times,** take notes and sketch the examples while thinking before you start with the code. |

#### Processing The Input Data

According to the assignment, we expect **four** lines containing different **integers** to be passed. Examining the provided parameters, we can use the **Number** type, as it is suitable for the expected values. We simultaneously **read** the input data and **parse** the string value to the selected data type for the **integer**.



Examining the expected output, we can create variables that contain the different output data types, to avoid using the so-called **"magic strings"** in the code.



#### Calculations

After reading the input data, we can now start writing the logic for calculating the result. Let's first calculate the **start time** of the exam **in minutes** for easier and more accurate comparison:



Let's also calculate the **student arrival time** using the same logic:



What remains is to calculate the difference between the two times, to determine **when** and **what time compared to the exam time** the student arrived at:



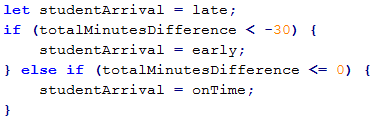
Our next step is to do the required **checks and calculations**, and finally, we will print the output. Let's separate the code into **two** parts.

* First, let's show when the student arrived – were they **early**, **late** or **on time**. To do that, we will use an **if-else** statement.
* After that, we will show the **time difference**, if the student arrives at a **different time** compared to the **exam starting time**.

To spare one additional check (**else**), we can, by default, assume that the student was late.

After that, according to the condition, we will check whether the difference in times is **more than 30 minutes**. If this is true, we assume that the student is **early**. If we do not match the first condition, we need to check if **the difference is less than or equal to zero (**<= 0**)**, by which we are checking the condition whether the student arrived within the range of **0 to 30 minutes** before the exam.

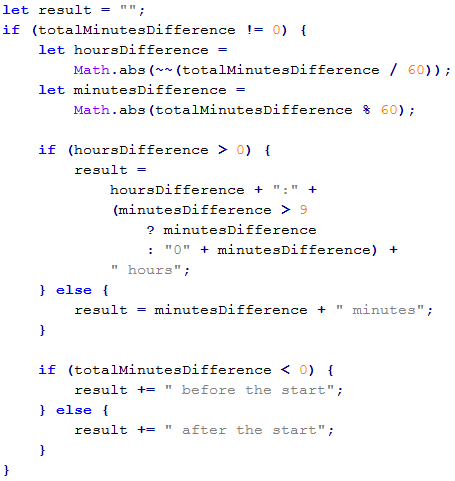
In all other cases, we assume that the student **was late**, which we set as **default**, and no additional check is needed:



Finally, we need to understand and print **what is the time difference between exam start time and student arrival time**, as well as whether this time difference indicates the time of arrival **before or after the exam start**.

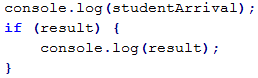
We check whether the time difference is **more than** one hour, to print hours and minutes in the required **format**, or **less than** one hour, to print **only minutes** as a format and description.

We also need to do one more check – whether the time of student's arrival is **before** or **after** the exam start time.



#### Printing The Result

Finally, what remains is to print the result on the console. According to the requirements, if the student arrived right on time (**not even a minute difference**), we do not need to print a second result. This is why we apply the following **condition**:



Actually, for the task, printing the result **on the console** can be done at a much earlier stage – during the calculations. This, however, is not a very good practice. **Why?** Let's examine the idea that our code is not 10 lines, but 100 or 1000! One day, printing the result will not be done on the console, but will be written in a **file** or displayed as a **web application**. Then, how many places in the code you will make changes at, due to such a correction? Are you sure you won't miss some places?

|  |  |
| --- | --- |
|  | Always consider the code that contains logical calculations as a separate part, different from the part that processes the input and output data. It has to be able to work regardless of how the data is passed to it and where the result will be displayed. |

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/932#0>.

## Problem: Trip

It is strange, but most people start planning their vacations well in advance. A young programmer from Bulgaria has a **certain budget** and spare time in a particular **season**.

Write a program that accepts **as input the budget and season** and **as output** displays programmer's **vacation place** and **the amount of money they will spend**.

**The budget determines the destination, and the season determines what amount of the budget will be spent**. If the season is summer, the programmer will go camping, if it is winter – he will stay in a hotel. If it is in Europe, regardless of the season, the programmer will stay in a hotel. Each camp or hotel, according to the destination, has its price, which corresponds to a particular **percentage of the budget**:

* If **100 BGN or less** – somewhere in **Bulgaria**.
  + **Summer** – **30%** of the budget.
  + **Winter** – **70%** of the budget.
* If **1000 BGN or less** – somewhere in the **Balkans**.
  + **Summer** – **40%** of the budget.
  + **Winter** – **80%** of the budget.
* If **more than 1000 BGN** – somewhere in **Europe**.
  + Upon traveling in Europe, regardless of the season, the programmer will spend **90% of the budget**.

### Input Data

The input data will be read from the console and will consist of **two lines** (arguments):

* The **first** line (argument) holds **the budget** – a **real number** in the range [**10.00 … 5000.00**].
* The **second** line (argument) holds **one** of two possible seasons that are "**summer**" or "**winter**".

### Output Data

**Two lines** must be printed on the console.

* On the **first** line – "**Somewhere in {destination}**" among "**Bulgaria**", "**Balkans**" and "**Europe**".
* On the **second** line – "{**Vacation type**} – {**Amount spent**}".
  + The **Vacation** can be in a "**Camp**" or "**Hotel**".
  + The **Amount** must be **rounded up to the second digit after the decimal point**.

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 50 summer | Somewhere in Bulgaria Camp - 15.00 |
| 75 winter | Somewhere in Bulgaria Hotel - 52.50 |
| 312 summer | Somewhere in Balkans Camp - 124.80 |
| 678.53 winter | Somewhere in Balkans Hotel - 542.82 |
| 1500 summer | Somewhere in Europe Hotel - 1350.00 |

### Hints and Guidelines

Typically, as for the other tasks, we can separate the solution into the following parts:

* Reading the input data
* Doing calculations
* Printing the result

#### Processing The Input Data

While reading carefully the requirements, we understand that we expect **two** parameters of input data. Our first parameter is a **real number**, for which we need to pick an appropriate variable type. For a higher level of calculation accuracy, we can pick **Number** as a variable for the budget and – **String** for the season.



|  |  |
| --- | --- |
|  | Always take into consideration what **value type** is passed in the input data, as well as what type these need to be converted to, for the program conditions to work properly! |

#### Calculations

Let's create and initialize the variables needed for applying the logic and calculations:



Similar to the example in the previous task, we can initialize variables with some of the output results, to spare additional initialization.

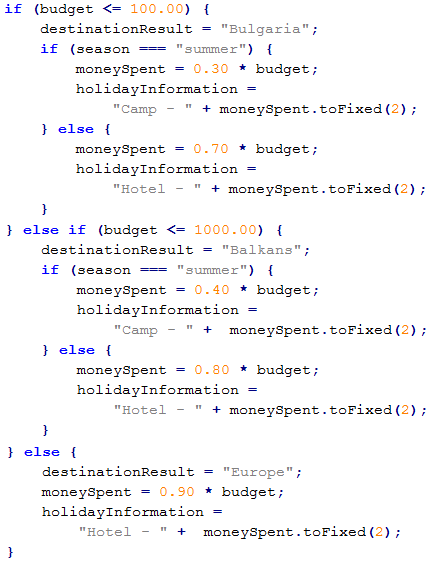
When examining once again the problem requirements, we notice that the main distribution of where the vacation will take place is determined by the **value of the budget**, i.e. our main logic is divided into two cases:

* If the budget is **less than** a particular value.
* If it is **less than** another value or is **more than** the specified border value.

Based on the way we arrange the logical scheme (the order in which we will check the border values), we will have more or fewer conditions in the solution. **Think why!**

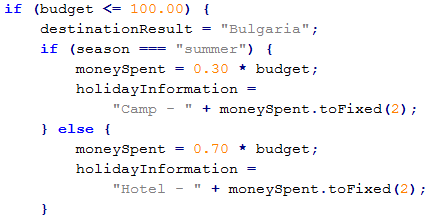
After that, we need to apply a condition to check the value of the **season**. Based on it, we will determine what percentage of the budget will be spent, as well as where the programmer will stay – in a **hotel** or a **camp**.

This is a sample code that may be used to implement the above idea:



We can optimize the conditions checking by assigning a **default value** and then checking one variant less. **This saves one logical step**.

For example, this block:



can be shortened like this:



#### Printing The Result

What remains is to display the calculated result on the console:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/932#1>.

## Problem: Operations

Write a program that reads **two integers (n1 and n2)** and an **operator** that performs a particular mathematical operation with them. Possible operations are: **summing up** (**+**), **subtraction** (**-**), **multiplying** (**\***), **division** (**/**) and **modular division** (**%**). Upon summing up, subtracting, and multiplying, the console must print the result and display whether it is an **even** or an **odd** number. Upon regular division – **just the result**, and upon modular division – **the remainder**. You need to take into consideration the fact that **the divisor can be equal to zero** (**= 0**) and dividing by zero is not possible. In this case, a **special notification** must be printed.

### Input Data

**3 arguments** are passed to the function:

* **N1** – **integer** within the range [**0 … 40 000**].
* **N2** – **integer** within the range [**0 … 40 000**].
* **Operator** – **one character** among: "**+**", "**-**", "**\***", "**/**", "**%**".

### Output Data

Print the output as a **single line** on the console:

* If the operation is **summing up**, **subtraction** or **multiplying**:
  + **"{N1} {operator} {N2} = {output} – {even/odd}"**.
* If the operation is **division**:
  + **"{N1} / {N2} = {output}"** – the result is **formatted** up **to the second digit after the decimal point**.
* If the operation is **modular division**:
  + **"{N1} % {N2} = {remainder}"**.
* In case of **dividing by 0 (zero)**:
  + **"Cannot divide {N1} by zero"**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 123 12 / | 123 / 12 = 10.25 | 112 0 / | Cannot divide 112 by zero |
| 10 3 % | 10 % 3 = 1 | 10 0 % | Cannot divide 10 by zero |

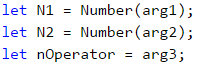
| **Input** | **Output** |
| --- | --- |
| 10 12 + | 10 + 12 = 22 - even |
| 10 1 - | 10 - 1 = 9 - odd |
| 7 3 \* | 7 \* 3 = 21 - odd |

### Hints and Guidelines

The problem is not complex, but there are a lot of code lines to write.

#### Processing The Input Data

Upon reading the requirements, we understand that we expect **three** parameters of input data. The first **two** parameters are **integers** (within the specified range), and the third one – **an arithmetical symbol**.



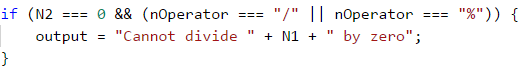
#### Calculations

Let's create and initialize the variables needed for the logic and calculations. In one variable we will store **the calculations output**, and in the other one, we will use it for the **final output** of the program.



When carefully reading the requirements, we understand that there are cases where we don't need to do **any** calculations, and simply display a result.

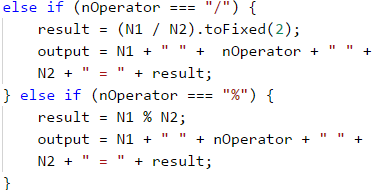
Therefore, we can first check if the second number is **0** (zero), as well as whether the operation is a **division** or a **modular division**, and then initialize the output.



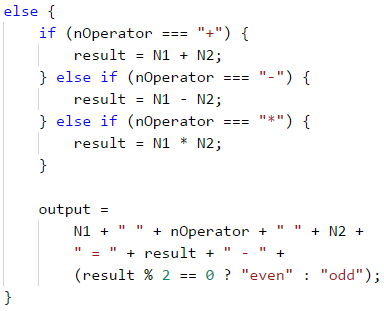
Let's place the output as a value upon initializing the **output** parameter. This way we can apply **only one condition** – whether it is needed to **recalculate** or **replace** this output.

Based on the approach that we choose, our next condition will be either a simple **else** or a single **if**. In the body of this condition, using additional conditions regarding the manner of calculating the output based on the passed operator, we can separate the logic based on the **structure** of the expected **output**.

From the requirements we can see that for **summing up** (**+**), **subtraction** (**-**) or **multiplying** (**\***) the expected output has the same structure: **"{n1} {operator} {n2} = {output} – {even/odd}"**, whereas for **division** (**/**) and **modular division** (**%**) the output has a different structure.

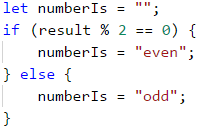


We finish the solution by applying conditions for summing up, subtraction, and multiplying:



For short and clear conditions, such as the above example for even and odd numbers, you can use a **ternary operator**. Let's examine the possibility to apply a condition **with** or **without** a ternary operator.

**Without using a ternary operator** the code is longer but easier to read:



**Upon using a ternary operator** the code is much shorter, but may require additional efforts to read and understand the logic:



#### Printing The Output

Finally, what remains is to print the calculated result on the console:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/932#2>.

## Problem: Match Tickets

**A group of football fans** decided to buy **tickets for Euro Cup 2016**. The tickets are sold in **two** price categories:

* **VIP** – **499.99** BGN (Bulgarian leva).
* **Normal** – **Normal** – **249.99** BGN (Bulgarian leva).

The football fans **have a shared budget**, and the **number of people** in the group determines what percentage of the budget will be **spent on transportation**:

* **1 to 4** – 75% of the budget
* **5 to 9** – 60% of the budget
* **10 to 24** – 50% of the budget
* **25 to 49** – 40% of the budget
* **50 or more** – 25% of the budget

**Write a program** that **calculates whether the money left in the budget** will be enough for the football fans to **buy tickets in the selected category**, as well as **how much money** they will **have left or be insufficient**.

### Input Data

The input data contains **exactly 3 lines** (arguments):

* The **first** line (argument) contains the **budget** – a real number within the range [**1 000.00 … 1 000 000.00**].
* The **second** line (argument) contains the **category** – "**VIP**" or "**Normal**".
* The **third** line (argument) contains the **number of people in the group** – an integer within the range [**1 … 200**].

### Output Data

**Print the following** on the console as **one line**:

* If the **budget is sufficient**:
  + "**Yes! You have {N} leva left.**" – where **N is the amount of remaining money** for the group.
* If the **budget is NOT sufficient**:
  + "**Not enough money! You need {M} leva.**" – where **M is the insufficient amount**.

**The amounts** must be **formatted up to the second digit after the decimal point**.

### Sample Input and Output

| **Input** | **Output** | **Explanations** |
| --- | --- | --- |
| 1000 Normal 1 | Yes! You have 0.01 leva left. | **1 person: 75%** of the budget is spent on **transportation**. **Remaining amount:** 1000 – 750 = **250**. Category **Normal**: the ticket **price is 249.99 \* 1 = 249.99** 249.99 < 250: **the person will have** 250 – 249.99 = **0.01** money left |

| **Input** | **Output** | **Explanations** |
| --- | --- | --- |
| 30000 VIP 49 | Not enough money! You need 6499.51 leva. | **49 people: 40%** of the budget are spent on **transportation**. Remaining amount: 30000 - 12000 = 18000. Category **VIP**: the ticket **costs** 499.99 \* 49. **24499.510000000002** < 18000. **The amount is not enough** 24499.51 - 18000 = **6499.51** |

### Hints and Guidelines

We will read the input data and perform the calculations described in the task requirements, to check if the money will be sufficient.

#### Processing The Input Data

Let's read carefully the requirements and examine what we expect to take as **input data**, what is expected to **return as a result**, as well as what are the **main steps** for solving the problem.

For a start, let's process and save the input data in **appropriate variables**:



#### Calculations

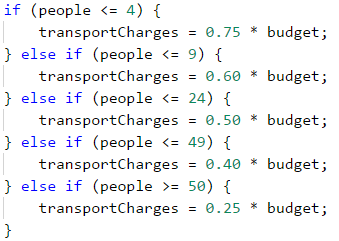
Let's create and initialize the variables needed for doing the calculations:



Let's review the requirements once again. We need to perform **two** different block calculations.

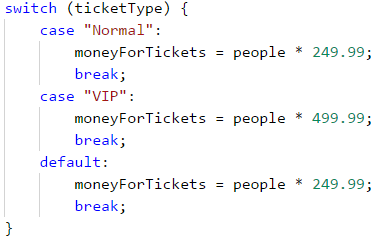
By the first set of calculations, we must understand what part of the budget has to be spent on **transportation**. You will notice that the logic for doing these calculations only depends on the **number of people in the group**. Therefore, we will do a logical breakdown according to the number of football fans.

We will use a conditional statement – a sequence of **if-else** blocks.



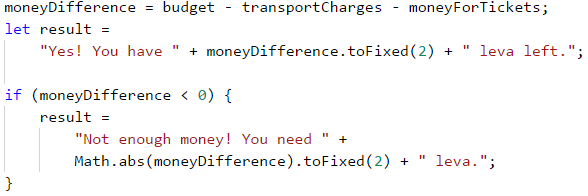
By the second set of calculations, we need to find out what amount will be needed for **purchasing tickets** for the group. According to the requirements, this only depends on the type of tickets that we need to buy.

Let's use a **switch-case** conditional statement.



Once we have calculated the **transportation costs** and **ticket costs**, what remains is to calculate the final result and understand if the group of football fans will **attend** Euro Cup 2016 or **not**, by the provided the available parameters.

For the output, to spare one **else condition** in the construction, we will assume that the group can, by default, attend Euro Cup 2016.



#### Printing The Result

Finally, we need to display the calculated result on the console.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/932#3>.

## Problem: Hotel Room

A hotel offers **two types of rooms**: **studio and apartment**.

Write a program that calculates **the price of the whole stay for a studio and apartment**. **Prices** depend on the **month** of the stay:

| **May and October** | **June and September** | **July and August** |
| --- | --- | --- |
| Studio – **50** BGN/per night | Studio – **75.20** BGN/per night | Studio – **76** BGN/per night |
| Apartment – **65** BGN/per night | Apartment – **68.70** BGN/per night | Apartment – **77** BGN/per night |

The following **discounts** are also offered:

* For a **studio**, in case of **more than 7** stays in **May and October**: **5% discount**.
* For a **studio**, in case of **more than 14** stays in **May and October**: **30% discount**.
* For a **studio**, in case of **more than 14** stays in **June and September**: **20% discount**.
* For an **apartment**, in case of **more than 14** stays, **no limitation regarding the month: 10% discount**.

### Input Data

The input data contains **exactly two lines** (arguments):

* The **first** (argument) line contains the **month** – **May**, **June**, **July**, **August**, **September** or **October**.
* The **second** line (argument) is the **number of stays** – integer within the range [**0 … 200**].

### Output Data

In this problem, our currency will be **lv**, which is BGN (Bulgarian lev). **Print** the following **two lines** on the console:

* On the **first line**: "**Apartment: { price for the whole stay } lv.**"
* On the **second line**: "**Studio: { price for the whole stay } lv.**"

**The price for the whole stay must be formatted up to two symbols after the decimal point**.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| May 15 | Apartment: 877.50 lv. Studio: 525.00 lv. | In **May**, in case of more than **14 stays**, the discount for a **studio is 30%** (50 - 15 = 35), and for the **apartment is 10%** (65 - 6.5 = 58.5). The whole stay in the **apartment: 877.50** lv. The whole stay **in the studio: 525.00** lv. |

| **Input** | **Output** |
| --- | --- |
| June 14 | Apartment: 961.80 lv. Studio: 1052.80 lv |
| August 20 | Apartment: 1386.00 lv. Studio: 1520.00 lv. |

### Hints and Guidelines

We will read the input data and do the calculations according to the provided price list and the discount rules, and finally, print the result.

#### Processing The Input Data

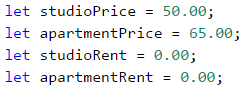
According to the task requirements, we expect two parameters, that contain the input data - the first parameter is **the month in which the stay is planned**, and the second - **the number of stays**.

Let's process and store the input data in the appropriate parameters:



#### Calculations

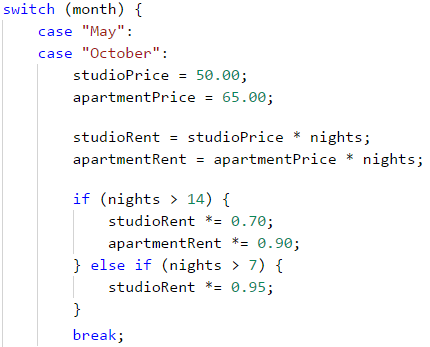
Now let's create and initialize the variables needed for the calculations:



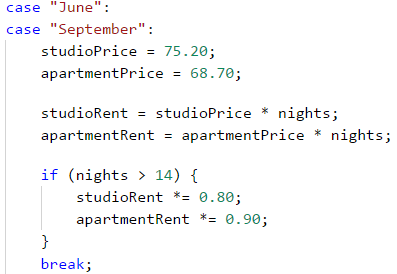
When doing an additional analysis of the requirements, we understand that our main logic depends on what **month** is passed and what is the number of **stays**.

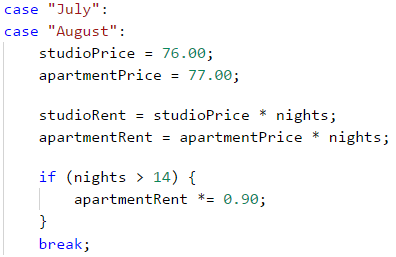
In general, there are different approaches and ways to apply the above conditions, but let's examine a basic **switch-case** conditional statement, as in the individual **case blocks** we will use **if** and **if-else** conditional statements.

Let's start with the first group of months: **May** and **October**. For these two months, **the price for a stay is the same** for both types of accommodation – a **studio** or an **apartment**. Therefore, the only thing that remains is to apply an internal condition regarding the **number of stays** and recalculate **the relevant price** (if needed):

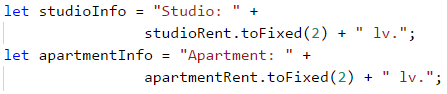


To some extent, the **logic** and **calculations** will be **identical** for the following months:





After calculating the relevant prices and the total amount for the stay, now let's prepare the formatted result. Before that, we should store it in our output **parameters** – **studioInfo** and **apartmentInfo**:



To calculate the output parameters, we will use the **.toFixed(Number)** **method**. This method **rounds the decimal** number up to a **specified number of characters** after the decimal point. In our case, we will round the decimal number up to **2 digits** after the decimal point.

#### Printing The Result

Finally, what remains is to print the calculated results on the console.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/932#4>.

# Chapter 5.1. Loops (Repetitions)

In this chapter, we will introduce the **repeat blocks of commands**, also known as "**loops**". We will write a set of loops, using the **for** operator in its simplest form. Finally, we will solve a few practical problems that require repetition of commands, using loops.

## Repeating Code Blocks (For Loops)

In programming, we often need **to execute a block of commands multiple times**. To do that we use **loops**. Let's look at one example of a **for loop**, which will loop through the numbers from 1 to 10 and print each one:

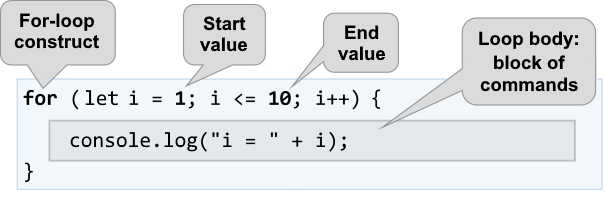
for (let i = 1; i <= 10; i++) {

console.log("i = " + i);

}

The loop starts with the **operator for** and loops through all values of a given variable in a given range, for example, all numbers from 1 to 10 included. For each value, it executes a sequence of commands.

Upon declaring the loop you can specify a **start value** and an **end value**. **The loop's body** is usually in curly braces **{ }** and contains one or more commands. The figure below shows the structure of a **for loop**:



A **for loop** often circles **1** to **n** times (for example 1 to 10). The goal is to **loop** through the numbers 1, 2, 3, …, n sequentially and after each pass through the loop to **perform a certain action**. In the example above the variable **i** holds values from 1 to 10 and the current value is printed in the loop's body. The loop repeats 10 times and each of these repetitions is called an "**iteration**".

### Problem: Numbers from 1 to 100

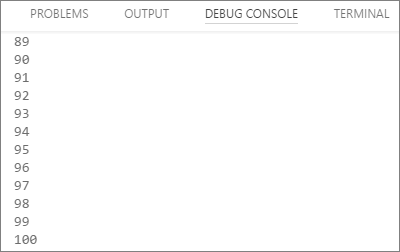
Write a program that **prints the numbers from 1 to 100**. The program does not accept input and prints the numbers from 1 to 100 sequentially, each on a separate line.

#### Hints and Guidelines

We can solve this problem with a **for loop** which will pass through the numbers from 1 to 100 using the variable **i** and will print the numbers in the loop's body:



**Start** the program with [**Ctrl+F5**] and **test** it:



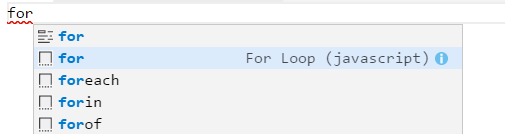
#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#0>.

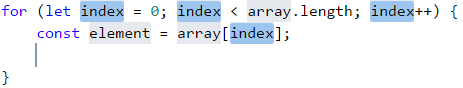
You should get **100 points** (completely correct solution).

## Code Snippet for a For loop in Visual Studio Code

While programming, we often need to use loops, dozens of times a day. That's why in most development environments (IDE) there are **code snippets** for writing loops. An example of such a snippet is the **snippet for a for loop in Visual Studio Code**. Write **for** in the JavaScript code editor in Visual Studio Code and **hit** [**Tab**]:



**VS Code** will open a snippet and write a complete **for loop**:



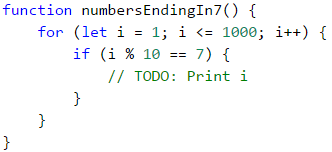
**Try it yourself**, so you can master the skill of using the code snippet for the **for loops** in Visual Studio Code.

### Problem: Numbers Ending in 7

Write a program that finds all numbers ending in 7 in the range [**1 … 1000**].

#### Hints and Guidelines

We can solve this problem by combining a **for loop** that loops through the numbers from 1 to 1000 and a **conditional statement** that checks if the number ends in 7. Of course, there are other options, too, but let's solve the problem using a **for loop + conditional statement**:



#### Testing in The Judge System

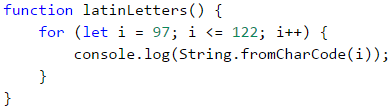
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#1>.

### Problem: Latin Letters

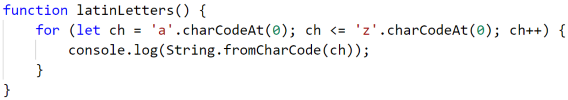
Write a program that prints the letters from the alphabet: **a, b, c, …, z**.

#### Hints and Guidelines

We can solve this problem using a **for loop**, that loops through all letters' code numbers. Keep in mind that the code of the letter (the serial number in the [Unicode list of letters and characters](https://unicode-table.com/en/)) ‘a’ is 97, the code of the letter ‘b’ is 98 and so on, and the code of the letter ‘z’ is 122. The transition from the number of the character to the letter itself is done with the help of the function String.fromCharCode(x). Here's the example:



If we want to make the code easier to read, we can write the following:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#2>.

### Problem: Sum Numbers

Write a program that **sums n given integer numbers**.

* The first line of the input holds the number of integers **n**.
* Each of the following **n** lines holds a number to sum.
* Sum up the numbers and finally print the sum.

#### Sample Input and Output

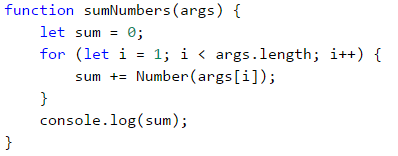
| **Input** | **Output** |
| --- | --- |
| 2 10 20 | 30 |
| 3 -10 -20 -30 | -60 |
| 4 45 -20 7 11 | 43 |
| 1 999 | 999 |
| 0 | 0 |

#### Hints and Guidelines

We can solve the problem by summing up numbers in the following way:

* Read the input number **n**.
* We start with a sum that initially equals zero **sum = 0**.
* We run a loop from 1 to **args.length**. On each step of the loop, we read the number **args[i]** and add it to the **sum**.
* Finally, we print the calculated **sum**.

Here's the source code for the solution:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#3>.

### Problem: Max Number

Write a program that reads **n integer numbers** (**n** > 0) and finds **the biggest** among them. The first line of the input specifies the number of integers **n**. The next **n** lines hold the numbers, one per line. Examples:

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 2 100 99 | 100 |
| 3 -10 20 -30 | 20 |
| 4 45 -20 7 99 | 99 |
| 1 999 | 999 |
| 2 -1 -2 | -1 |

#### Hints and Guidelines

First, we read one number **n** (the number of integers that will be entered). We assign an initial neutral value to the current maximum **max**, for example **-1000000** (or **Number.NEGATIVE\_INFINITY**). Using a **for loop** that iterates **n times** (**n = args[0]**), we read one integer number **num** on each iteration. If the current number **num** is higher than the current maximum **max**, we assign the value of the **num** to the variable **max**. Finally, we have the highest number's value stored in **max**. We print the number on the console.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#4>.

### Problem: Min Number

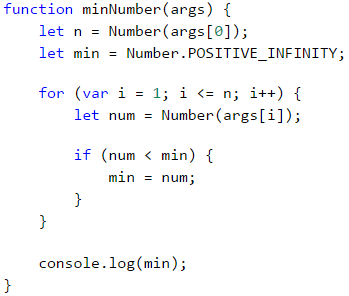
Write a program that reads **n integer numbers** (**n** > 0) and finds **the smallest** among them. First, read the number of integers **n**, then **n** numbers, one per line.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 2 100 99 | 99 |
| 3 -10 20 -30 | -30 |
| 4 45 -20 7 99 | -20 |

#### Hints and Guidelines

The problem is completely identical to the previous one, except this time we will use another neutral value for a start.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#5>.

### Problem: Left and Right Sum

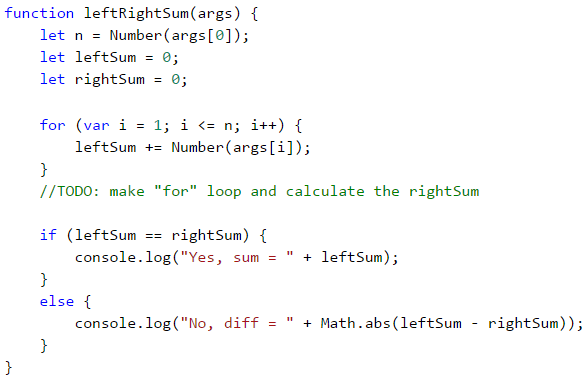
Write a program that reads **2 \* n integer numbers** and checks whether **the sum of the first n integers** (left sum) equals **the sum of the second group of n integers** (right sum). In case the sums are equal, print **"Yes" + the sum**, otherwise print **"No" + the difference**. The difference is calculated as a positive number (by absolute value). The format of the output must be identical to the one in the examples below.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 10 90 60 40 | Yes, sum = 100 | 2 90 9 50 50 | No, diff = 1 |

#### Hints and Guidelines

First, we read the number **n**, after that we enter the first **n** numbers (**the left** half) and calculate their sum. We will then proceed to read more **n** numbers (**the right** half) and sum them up. We calculate **the difference** between the sums by absolute value: **Math.abs(leftSum - rightSum)**. If the difference is **0**, print **"Yes" + the sum**, otherwise print **"No" + the difference**.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#6>.

### Problem: Odd Even Sum

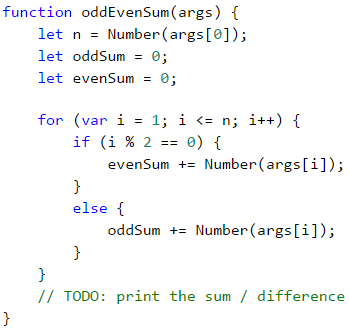
Write a program that reads **n integer numbers** and checks whether **the sum of the numbers on even positions** equals **the sum of the numbers on odd positions**. In case the sums are equal, print **"Yes" + the sum**, otherwise print **"No" + the difference**. The difference is calculated by absolute value. The format of the output must be identical to the one in the examples below.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 4 10 50 60 20 | Yes Sum = 70 |
| 4 3 5 1 -2 | No Diff = 1 |
| 3 5 8 1 | No Diff = 2 |

#### Hints and Guidelines

The program reads the numbers one by one and calculates the two **sums** (the sum of numbers on **even** positions and the sum of numbers on **odd** positions). We calculate the absolute value of the difference, like in the previous problem, and print the result (**"Yes" + the sum** in case the difference is 0 or **"No" + the difference** in any other case).



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#7>.

### Problem: Vowels Sum

Write a program that reads a **text** (string), calculates and prints **the sum of the vowels' value** according to the table below:

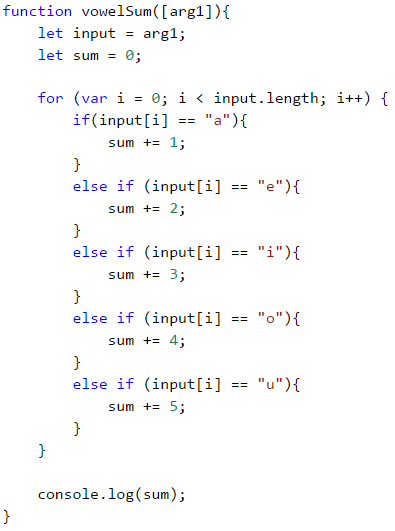
| **a** | **e** | **i** | **o** | **u** |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| hello | 6 (e+o = 2+4 = 6) | bamboo | 9 (a+o+o = 1+4+4 = 9) |
| hi | 3 (i = 3) | beer | 4 (e+e = 2+2 = 4) |

#### Hints and Guidelines

We read the text input **arg1**, then create a sum that equals zero and run a loop from **0** to **input.length** (the text's length). Check whether each letter **input[i]** is a vowel and add its value to the sum.

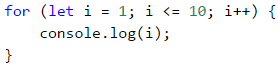


#### Testing in The Judge System

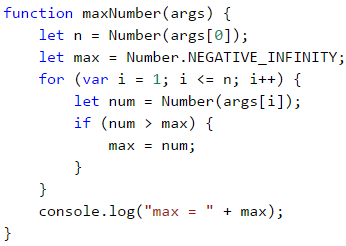
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#8>.

## What Have We Learned from This Chapter?

We can repeat a block of code using a **for loop**:



We can perform various mathematical operations:



## Problems: Loops (Repetitions)

Now that we got acquainted with the loops, it's time **to consolidate our knowledge in practice**, and as you know, this is done with a lot of code writing. Let's solve several problems for exercise:

### Problem: Half Sum Element

Write a program that reads **n integer numbers** and checks whether there is a number that equals the sum of all the rest among them. If there is such an element, print **"Yes" + the element's value**, otherwise - **"No" + the difference between the largest element and the sum of the rest** (by absolute value).

#### Sample Input and Output

| **Input** | **Output** | **Comment** |
| --- | --- | --- |
| 7 3 4 1 1 2 12 1 | Yes Sum = 12 | 3 + 4 + 1 + 2 + 1 + 1 = 12 |
| 4 6 1 2 3 | Yes Sum = 6 | 1 + 2 + 3 = 6 |
| 3 1 1 10 | No Diff = 8 | |10 - (1 + 1)| = 8 |
| 3 5 5 1 | No Diff = 1 | |5 - (5 + 1)| = 1 |
| 3 1 1 1 | No Diff = 1 | - |

#### Hints and Guidelines

Calculate **the sum** of all elements, find **the biggest** among them, and check the condition.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#9>.

### Problem: Odd / Even Position

Write a program that reads **n numbers** and calculates **the sum**, **the min** and **the max** value of the numbers on **even**, and **odd** positions (counting from 1). If no min / max element exists, print **"No"**.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 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 1 | OddSum=1, OddMin=1, OddMax=1, EvenSum=0, EvenMin=No, EvenMax=No | 0 | OddSum=0, OddMin=No, OddMax=No, EvenSum=0, EvenMin=No, EvenMax=No |
| 5 3 -2 8 11 -3 | OddSum=8, OddMin=-3, OddMax=8, EvenSum=9, EvenMin=-2, EvenMax=11 | 4 1.5 1.75 1.5 1.75 | OddSum=3, OddMin=1.5, OddMax=1.5, EvenSum=3.5, EvenMin=1.75, EvenMax=1.75 |
| 1 -5 | OddSum=-5, OddMin=-5, OddMax=-5, EvenSum=0, EvenMin=No, EvenMax=No | 3 -1 -2 -3 | OddSum=-4, OddMin=-3, OddMax=-1, EvenSum=-2, EvenMin=-2, EvenMax=-2 |

#### Hints and Guidelines

The problem combines some of the previous problems: finding **the min**, **the max** value, and **the sum**, as well as processing the elements on **even and odd positions**. Check them out.

In this problem, it's better to work with **fractions** (not integers). The sum, the minimum, and the maximum value are also fractions. We must use **a neutral start value** upon finding the minimum / maximum, for example **1000000.0** and **-1000000.0**. If the final result is the neutral value, print **“No”**.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#10>.

### Problem: Equal Pairs

There are **2 \* n numbers**. The first and the second number form **a pair**, so do the third and the fourth number, and so on. Each pair has **a value** – the sum of its numbers. Write a program that checks **whether all pairs have the same value**.

In case the value is the same, print **"Yes, value=…" + the value**, otherwise, print **the maximum difference** between two consecutive pairs in the following format - **"No, maxdiff=…" + the maximum difference**.

The input holds the number **n**, followed by **2\*n integers**, all of them on a separate line.

#### Sample Input and Output

| **Input** | **Output** | **Comment** |
| --- | --- | --- |
| 3 1 2 0 3 4 -1 | Yes, value=3 | values = {3, 3, 3} same values |
| 2 1 2 2 2 | No, maxdiff=1 | values = {3, 4} difference = {1} max. difference = 1 |
| 4 1 1 3 1 2 2 0 0 | No, maxdiff=4 | values = {2, 4, 4, 0} difference = {2, 0, 4} max. difference = 4 |
| 1 5 5 | Yes, value=10 | values = {10} only one value same values |
| 2 -1 0 0 -1 | Yes, value=-1 | values = {-1, -1} same values |
| 2 -1 2 0 -1 | No, maxdiff=2 | values = {1, -1} difference = {2} max. difference = 2 |

#### Hints and Guidelines

Read the numbers from the input **in pairs**. For each pair calculate its **sum**. As we read the number pairs from the input, for each pair, except for the first one, we must calculate **the difference with the previous one**. To do that, we need to store the sum of the previous pair in a separate variable. Finally, find **the biggest difference** between the two pairs. If it is **0**, print **“Yes”** + the value, otherwise - **“No”** + the difference.

#### Testing in The Judge System

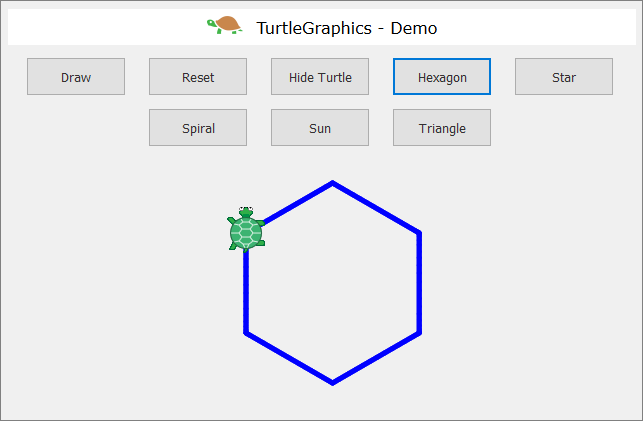
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/933#11>.

## Lab: Graphics and Web Applications

In this chapter, we learned about **loops** as a programming construction that allows us to repeat a particular action or a group of actions multiple times. Now let's play with them. To do that, we will draw several figures, which consist of a large number of repetitive graphic elements, and this time we will not do it on the console, but in a graphical environment, using "**turtle graphics**". It will be interesting. And it's not complicated at all. Try it!

### Problem: Drawing with a Turtle – a Graphical Application (GUI)

The purpose of the next exercise is to play with a **drawing library**, also known as **turtle graphics**. We will build a graphical application in which we will **draw different shapes**, moving our **“turtle”** on the screen through commands like “move 100 positions forward”, “turn 30 degrees to the right”, “move 50 positions more forward”. The application will look something like this:



First, let's check out **the drawing concept “Turtle Graphics”**. Take a look at the following sources:

* Definition of “turtle graphics”: <https://wiki.c2.com/?TurtleGraphics>
* Article on “turtle graphics” in Wikipedia: <https://en.wikipedia.org/wiki/Turtle_graphics>

We will build the application, using the following technologies:

* **HTML** language – to describe the user interface (drawing field and buttons).
* **JavaScript** code – to implement the actions of the buttons.
* JS library [**jQuery**](https://jquery.com/) – to facilitate access to the user's interface.
* JS library [**jQuerу-Turtle**](https://github.com/PencilCode/jquery-turtle) – to implement screen drawing with the mechanics of “turtle graphics”.

There are two options to load all libraries and resources for our web application **Turtle Graphics**:

* **Loading resources via CDN** (Content Delivery Network).

This option is suitable when we have a permanent internet connection. We need to make a standard HTML file (for example **index.html**) and write the following code in it:

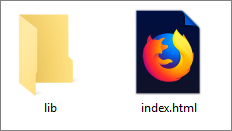


All the necessary resources will be loaded automatically when the file is started and we can directly write our **JavaScript** code.

If for some reason you don't have permanent access to the Internet, you can use the second option:

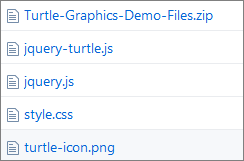
* **Loading Resources Locally**

You have to download all the necessary files and change a few lines in the HTML file. Start by creating a folder called **Turtle-Demo** and create the main **HTML file** and a subfolder for the necessary resources in it:



In the folder **"lib"** we need to put a few files that we can download from the book's online storage: <https://github.com/SoftUni/Programming-Basics-Book-JS-EN/tree/master/assets/chapter-5-1-assets>

For your convenience, we have put the files in an archive, easy to download **Turtle-Graphics-Demo-Files.zip**:



Let's take a look at each one of them:

**jquery.js (version 2.0.3)**

One of the most popular JavaScript libraries which offers **speed and functionality** when working with HTML user interface. It changes the way we write code and plan its structure. Check out the tool at: [https://jquery.com](https://jquery.com/)

**jquery-turtle.js (version 2.0.8)**

A plugin, written by **David Bau** for jQuery - **jQuery-turtle** which provides a set of features for the **graphic drawing** of the "turtle" type. Find detailed information and rules of use here: <https://github.com/davidbau/jquery-turtle>

**style.css**

A set of **design rules** in a separate file.

**turtle-icon.png**

A **raster graphic**, which we use for a better presentation of the application.

Once we have put all files in one folder, we need to change the network address of the resources in our **HTML** file - **index.html**:



After this change, each time you start the file, the browser will load the files locally from the **"lib"** folder.

Now we can proceed to the fun part - **writing the JavaScript code** for the web application. It will be located in the last pair of **script** tags in the HTML file (index.html):

<script> </script>

The code with the functions of the application will be relatively short (about 70-80 lines) and so we don't need to put it into a new separate file. The important thing is to place it correctly in our HTML file.

|  |  |
| --- | --- |
|  | It is recommended that all JavaScript files be placed at the end of the HTML document before the final **"body"** tag. This guarantees the faster loading of the page because we do not delay the rendering (processing) of its elements. |

Always **insert the library file first** [**jquery.js**], then the one with the plugin code [**jquery-turtle.js**]. Only then we will write our code because it's based on the first two files. If we try to change their places, we will get errors and our application will not work properly.

The library **"jQuery"** allows us to manipulate **HTML** elements, using valid design selectors (**CSS**). We have to apply a specific syntax:

$('#ID') or $('.Class')

We can use the **HTML** element's name, **ID**, or its **class**. The selectors are always strings of text, so they are enclosed in single or double-quotes. If the selector is the **ID** (a separate name for each element), in the beginning, we put a **sharp sign** (**#**). But if we have decided to select **by class** (one name for multiple elements), then we write a **dot**.

Adhering to the **jQuery-turtle** documentation, we have to initialize our object and set the basic characteristics of the "turtle". With the following code we will determine **the size** of the graph (**turtleScale**) and **the speed** of movement (**turtleSpeed**):

eval($.turtle());

$('#turtle').css('turtleScale', '2').css('turtleSpeed', '4');

Once we are done with the basis of our application, we have to write the functions for each button. To do that, we use the selectors (**ID**) of the objects that we have preset in the **HTML file**. We will share the code for the first three buttons to get you acquainted with the basic principles:

* **"Draw" Button**

We attach a function to the element with an **ID** (selector) **"justDraw"**, which will be activated the moment you click on the button:

$('#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);

}

});

First, **erase the graphs** with the function **cg()** (clear graphics) and **write** an elementary **loop** that repeats 4 times. We use the keyword **let** on purpose, to define the variable **index**. This way we guarantee the variable's autonomy for the specific loop and we can easily use the same name again.

On each iteration (repetition) we apply specific methods of movement with a certain value (turn to the left and move forward):

.lt(30) // rotate to the left with an argument 30

.fd(150) // move forward with an argument 150

Using the chaining technique saves additional code writing:

$('#turtle').pen('blue', '5').lt(30).fd(150).lt(120) ...

// which is a shorter version of the classic method:

$('#turtle').pen('blue', '5');

$('#turtle').lt(30);

$('#turtle').fd(150);

$('#turtle').lt(120);

$('#turtle').fd(150);

...

* **"Reset" Button**

Attach a function to the element with an **ID** (selector) **"reset"**, which will be activated at the moment of clicking:

$('#reset').click(function() {

window.location.reload();

});

With **window.location.reload()** we activate **reloading the window**, which resets its current state. It is important to note that the **location** is **a feature** of the object **window**, and **reload()** is **a method** of **location**.

* **"Hide Turtle" Button**

Attach a function to the element with an **ID** (selector) **"hide"**, which will be activated on clicking the button:

$('#hide').click(function() {

$('#turtle').toggle();

$(this).text(function(i, text) {

return text === "Hide Turtle" ? "Show Turtle" : "Hide Turtle";

});

});

We use a preset function from the library **jQuery - toggle()**. With its help, we can **hide and show** elements. Apart from that, we attach another function that **changes the button's text** when clicked. We have to use the keyword **this**. It plays an important role in the JavaScript language syntax and **our thoughts** we can replace it with **self/itself**. In this case, it is equal to the element **hide**, ie. **this = #hide**. We turn to the element itself and set a text changing function which is also activated each time the button is clicked.

Let's sum up the code we have written so far:

<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>

The only thing left to solve is the problem of automatically deleting the drawing field when a new button is clicked. We don't want the shapes to be drawn one over the other or to use the **Reset** button every time.

* **Function resetCanvas()**

function resetCanvas() {

cg();

home();

$('#turtle').css('turtleScale', '2').css('turtleSpeed', '4');

}

Apply the preset functions in David Bau's plugin to **erase all graph elements** (**cg()**) and **move** the turtle to its **starting position** (**home()**). Then we set the initial settings of the **turtle** element once again. The whole function **resetCanvas()** is added at the beginning of each new function that will be attached to the other buttons.

**Example:**

$('#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

}

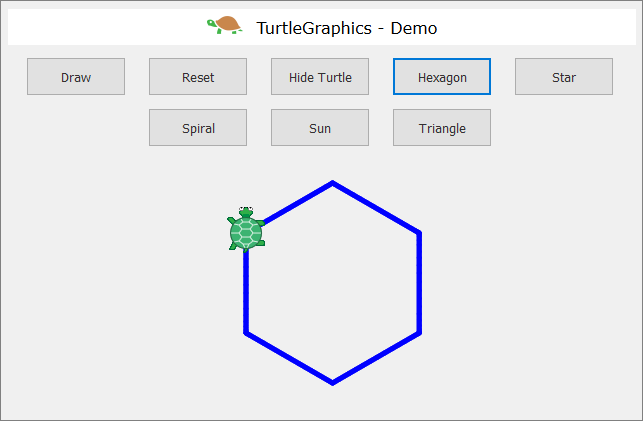
});

If you want, you can change the animation speed and the contour color by adding the new setting directly to the **click(...)** function and the function attached to it for each button:

$('#turtle').css('turtleSpeed', '6').pen('red', '5');

### Problem: \* Draw a Hexagon with The Turtle

Add a [**Hexagon**] button that draws a regular hexagon:



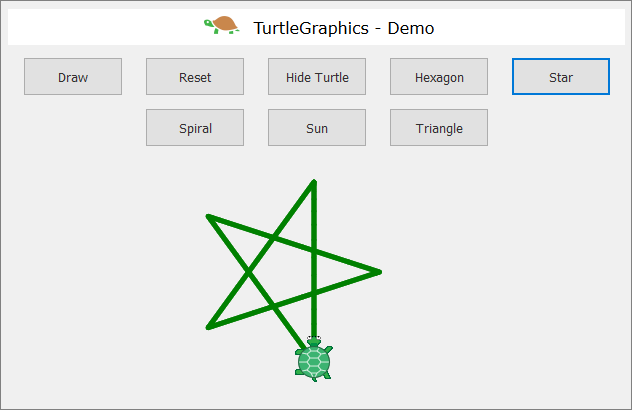
**Hint:**

Repeat 6 times the following in a loop:

* 60 degrees rotation.
* Move forward 90 steps.

### Problem: \* Draw a Star with The Turtle

Add a [**Star**] button that draws a star with 5 beams (**pentagram**), as shown in the figure below:



**Hint:**

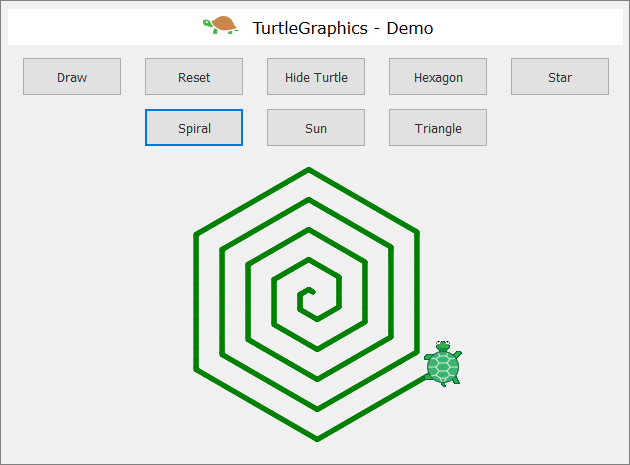
Change the color: **$("#turtle").pen("green", "5")**

Repeat 5 times the following in a loop:

* Move forward 180 steps.
* 144 degrees rotation.

### Problem: \* Draw a Spiral with The Turtle

Add a [**Spiral**] button that draws a spiral with 30 beams, as shown in the figure below:

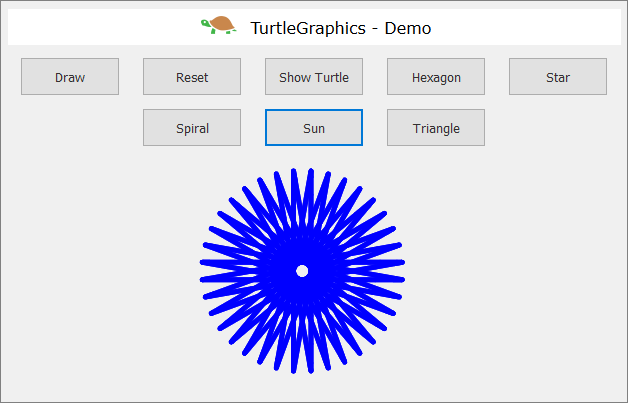


**Hint:**

Draw in a loop by moving forward and rotating. In each step, increase gradually the length of the forward step with 5 and rotate 60 degrees.

### Problem: \* Draw a Sun with The Turtle

Add a [**Sun**] button that draws a sun with 36 beams, as shown in the figure below:



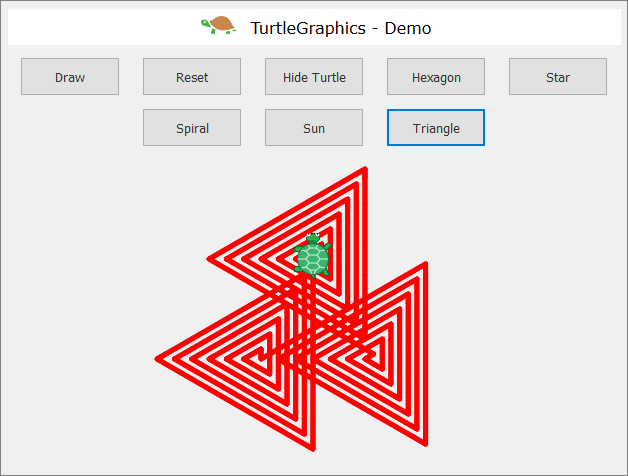
**Hint:**

Repeat the following 36 times in a loop:

* Move forward 200 steps.
* 170 degrees rotation.

### Problem: \* Draw a Spiral Triangles with The Turtle

Add a [**Spiral Triangle**] button that draws three triangles with 22 beams each, as shown in the figure below:



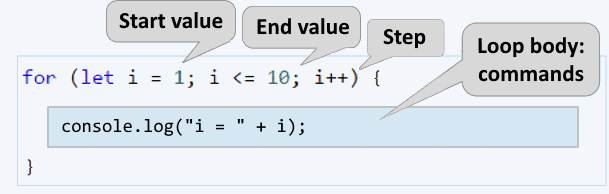
**Hint:**

Draw in a loop by moving forward and rotating. In each step, increase the length of the forward step by 10 and rotate 120 degrees. Repeat 3 times in another loop for the three triangles.

If you have difficulties with the problems above, ask for help in **SoftUni's discussion Reddit**: <https://www.reddit.com/r/softuni/>.

# Chapter 5.2. Loops – Exam Problems

In the previous chapter, we learned how to run a block of commands **more than once**. That's why we introduced a **for loop** and reviewed some of its main use cases. The purpose of this chapter is to improve our knowledge, by solving some more complex exam problems with loops, used for a practical exams. For some of them, we will show examples of comprehensive solutions, and for others, we will provide only guidance. Before we start, let's take another look at the **for loop** construction:



The **for loops** consist of:

* **Initialization expression**, in which the variable-counter (**let i**) is created and its initial value is set.
* **Condition expression** (**i <= 10**), executed once, before each loop iteration.
* **Increment expression** (**i++**) – expression executed after each iteration.
* **Body** of the loop - contains any block of source code

## Exam Problems

Let's solve some SoftUni exam problems with loops

## Problem: Histogram

**N integers** are given within the range of [**1 … 1000**]. A percentage of them **p1** are with values below 200, another percentage **p2** are with values from 200 to 399, percentage **p3** are with values from 400 to 599, a percentage **p4** are with values from 600 to 799 and the remaining percentage **p5** representing values above 800. Write a program that calculates and prints the percentages **p1**, **p2**, **p3**, **p4**, and **p5**.

**Example**: we are given n = **20** numbers: 53, 7, 56, 180, 450, 920, 12, 7, 150, 250, 680, 2, 600, 200, 800, 799, 199, 46, 128, 65 in the following distribution and visualization:

| **Group** | **Numbers** | **Number count** | **Persent** |
| --- | --- | --- | --- |
| < 200 | 53, 7, 56, 180, 12, 7, 150, 2, 199, 46, 128, 65 | 12 | p1 = 12 / 20 \* 100 = 60.00% |
| 200 … 399 | 250, 200 | 2 | p2 = 2 / 20 \* 100 = 10.00% |
| 400 … 599 | 450 | 1 | p3 = 1 / 20 \* 100 = 5.00% |
| 600 … 799 | 680, 600, 799 | 3 | p4 = 3 / 20 \* 100 = 15.00% |
| ≥ 800 | 920, 800 | 2 | p5 = 2 / 20 \* 100 = 10.00% |

### Input Data

The first line (argument) of the input is an integer **n** (1 ≤ **n** ≤ 1000), representing the count of lines with numbers that will be passed. On the following **n lines (arguments)** will be given **one integer** within the range of [**1 … 1000**] – based on these numbers the histogram should be calculated.

### Output Data

Print on the console **histogram that consists of 5 lines**, each of them containing a number between 0% and 100%, formatted with precision two digits after the decimal point (for example 25.00%, 66.67%, 57.14%).

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| **3** 1 2 999 | 66.67% 0.00% 0.00% 0.00% 33.33% | **4** 53 7 56 999 | 75.00% 0.00% 0.00% 0.00% 25.00% |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| **7** 800 801 250 199 399 599 799 | 14.29% 28.57% 14.29% 14.29% 28.57% | **9** 367 99 200 799 999 333 555 111 9 | 33.33% 33.33% 11.11% 11.11% 11.11% |

| **Input** | **Output** |
| --- | --- |
| **14** 53 7 56 180 450 920 12 7 150 250 680 2 600 200 | 57.14% 14.29% 7.14% 14.29% 7.14% |

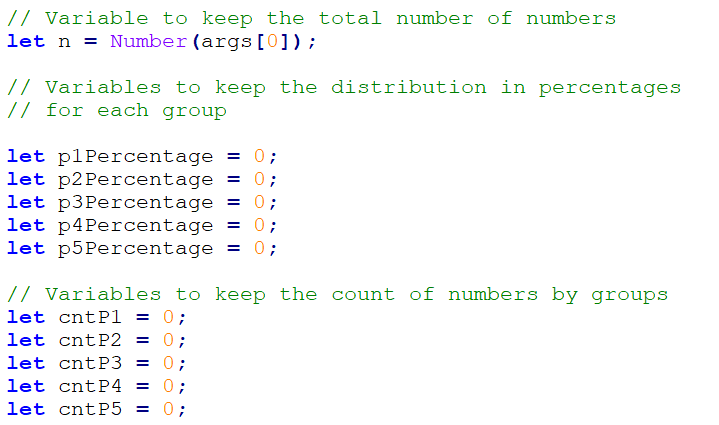
### Hints and Guidelines

The program that solves this problem could be divided into three parts:

* **Reading the input data** – the current problem includes a reading of the number **n**, followed by **n integers**, each on a single line.
* **Processing the input data** – in this case, that means allocating the numbers into groups and calculating the percentage breakdown by groups.
* **Print the output** – print the histogram in the specified format.

#### Reading The Input Data

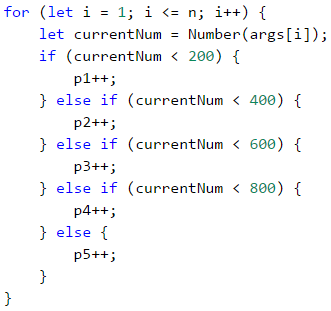
Before we proceed to the actual reading of the input data, we have to **declare the variables**, in which the input data will be stored:



In the variable **n**, we will store the count of numbers that we will need to read. Additionally, we have to declare the variables **p1**, **p2**, etc., in which we will store the count of numbers for each corresponding group. Once the variables have been declared we can proceed with input data processing.

#### Processing The Input Data

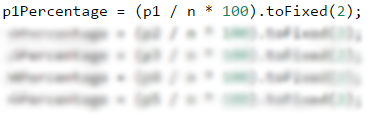
To be able to read and distribute each number to its respective group, a **for loop** from **0** to **n** (count of numbers) will be used. Each loop iteration will read and distribute **one single** number (**currentNum**) to its respective group. To define, if a number belongs to a certain group we have to, **perform a check in the respective range**. Once the check returns true - we increase the count of numbers in this group (**p1**, **p2**, etc.) by 1:



After determining the count of numbers in each group we can move on to the main objective and calculate the percentages. For the calculation we will use the following formula:

**(group percentage) = (count of numbers in a group) / (count of all numbers) \* 100**

This formula in the source code would look as:



According to the task assignment, the percentages have to be **formatted with precision two digits after the decimal point**. Considering this, **.toFixed(...)** method should be added to the formula and the first variable would look like this:

p1Percentage = (p1 / n \* 100).toFixed(2);

//Add the formula for the rest of the variables

To make it even clearer, let's take a look at the following example:

| **Input** | **Output** |
| --- | --- |
| **3** 1 2 999 | 66.67% 0.00% 0.00% 0.00% 33.33% |

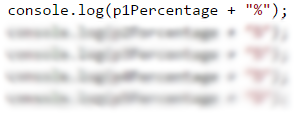
In this case **n = 3**. For this loop we have:

* **i = 0** - we read the number 1, which is less than 200 and falls into the first group (**p1**), we increase the corresponding group counter by 1.
* **i = 1** – we read the number 2, which again falls into the first group (**p1**) and we increase the same group counter by 1.
* **i = 2** – we read the number 999, which falls into the last group (**p5**) because it's bigger than 800, and we increase the corresponding group counter by 1.

After reading the numbers, we have in group **p1** two numbers and in group **p5** one number. For the rest of the groups, we have **no numbers**. The percentage of each group is calculated by applying the above-mentioned formula.

#### Printing The Output

All we have to do at this point is to print the final results:



### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#0>.

## Problem: Smart Lilly

Lily is **N years old** already. For each **birthday** she receives a present. For each **odd** birthday (1, 3, 5, …, n) she receives **toys**, and for the **even** birthdays (2, 4, 6, …, n) she receives **money**. For her **second birthday**, she received **10.00 USD**, and the **amount is increased by 10.00 USD for every subsequent even birthday** (2 -> 10, 4 -> 20, 6 -> 30, etc.). Over the years Lily has secretly saved her money. In the years when Lily received money, her brother each time took 1 USD. Lily **sold the toys**, she got over the years, **each one for P USD**, and then added the sum to her savings. With her savings, she wanted to **buy herself a washing machine for X USD**. Write a program, that calculates the **amount of Lily's savings** and is this sum enough **for buying a washing machine**.

### Input Data

The program receives **3 numbers** (arguments), entered by the user on separate lines:

* Lily's **age** – **integer** in the range of [**1 … 77**].
* **Washing machine price** – a number in the range of [**1.00 … 10 000.00**].
* **Single toy price** – **integer** in the range of [**0 … 40**].

### Output Data

In a single line print on the console:

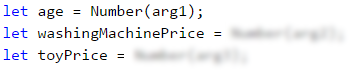
* If Lily has enough money to make the purchase:
  + "**Yes! {N}**" – where **N** is the money left after the purchase
* If Lily's savings are not enough for making a purchase:
  + "**No! {M}**" – where **M** is the insufficiency amount
* The numbers **N** and **M** must be **formatted with precision two digits after the decimal point**.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 10 170.00 6 | Yes! 5.00 | **First birthday** receives **toy**; **2nd** -> **10 USD**; 3rd -> toy; **4th** -> 10 + 10 = **20 USD**; 5th -> toy; **6th** -> 20 + 10 = **30 USD**; 7th -> toy; **8th** -> 30 + 10 = **40 USD**; 9th -> toy; **10th** -> 40 + 10 = **50 USD**. **She has saved** -> 10 + 20 + 30 + 40 + 50 = **150 USD**. She sold**5 toys 6 USD each = 30 USD**. **Her brother took 1 USD 5 times**. **Remaining amount of money** -> 150 + 30 – 5 = **175 USD**. **175 >= 170** (the price of the washing machine) **she was able** to buy the washing machine and she was **left** 175-170 = **5 USD**. |
| 21 1570.98 3 | No! 997.98 | **She saved 550 USD.** **She sold** **11 toys** **3 USD each.** = **33 USD**.Her brother **has taken 1 USD each year for 10 years** = **10 USD**. **Remaining amount of money** 550 + 33 – 10 = **573 USD** **573 < 1570.98** – **She didn't manage** to buy a washing machine. **The insufficient amount of money is** 1570.98–573 = **997.98 USD** |

### Hints and Guidelines

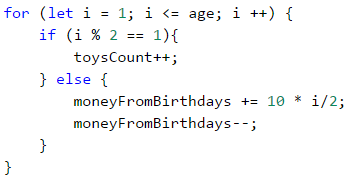
Similar to the previous problem, the solution could be divided into three parts – **reading** the input data, **processing** the input data, and **printing the output**.



Again, we should start with choosing appropriate variable names. For Lily's age(**age**), for the washing machine price (**washingMachinePrice**) and the single toy price (**toyPrice**). In the above code we **declare** and **initialized** (assign value to a variable) also variables for the toys count(**toysCount**), and money from birthdays (**moneyFromBirthdays**):



We use a **for loop** to iterate through every Lily's birthday. If the leading variable is an **odd number**, we increase the count of **toys**. We can do the parity check using **division with the remainder** (**%**) **by 2** – if the remainder is equal to 0, the number is **even**, and if the remainder is equal to 1 - **odd**. Inversely, if the leading variable is an **even number**, that would mean, that Lily has **received money** so we have to add this amount of money to her savings. Then we **Increase** the value of the variable **moneyFromBirthdays**, i.e. **we increase by 10** the sum, that she will receive for her next birthday. At the same time, we **subtract 1 USD** - the money taken from her brother. To use the decrement operator we have to add two minus signs after the last sign of the variable (**moneyFromBirthdays--**):



Probably you will encounter some difficulties calculating the amount of birthday money if you let the bonus money be added in the following way:

moneyFromBirthdays += 10;

The final result would be **10 x 5 = 50**, while our goal is to have **10 + 20 + 30 + 40 + 50 = 150**. The problem could be solved by declaring an additional variable(**bonusMoney**):

bonusMoney += 10;

moneyFromBirthdays += bonusMoney;

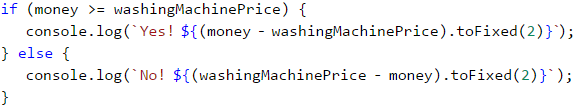
Or we can include the value of the variable **i**, which **counts the loops** and then divide it by **2**:

moneyFromBirthdays += 10 \* i/2;

Then we have to add the money received from the sold toys to Lily's savings.



At this point what is left is to print the results considering the required formating i.e. the sum should be **formatted with precision to the second digit after a decimal point**:



To avoid declaring additional variables we can use a template literal - **${expression}**. It's a text literal with a specific sequence of characters allowing embedding of expressions. Using it the calculation can be performed and the result to be directly included in the text string.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#1>.

## Problem: Back to The Past

Ivan is **18 years old** and just received an inheritance, consisting of **X amount of money** and a **time machine**. He decides **to return to the year 1800**, but he is not certain **if the money** would be **enough** for him, to live at that time without working. Write a **program, that calculates** if Ivan **would have enough money**, to live without working **until a given year (inclusively)**. Assuming that, for **every even** (1800, 1802 etc.) year he **would spend 12 000 USD**. For **every odd year** (1801,1803 etc.) he would spend **12 000 + 50 \* [the age he would have reached in the given year]**.

### Input Data

The program receives **2 numbers (arguments)**, entered by the user on separate lines

* **Inherited money** – a number in the range of [**1.00 … 1 000 000.00**].
* **The year, until he would have to live in the past (inclusively)** – an integer in the range of [**1801 … 1900**].

### Output Data

**Print** on the console **1 line**. **The sum** must be **formatted** with precision up to **two digits after the decimal point**:

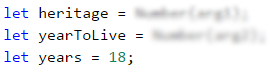
* If **the money is enough**:
  + „**Yes! He will live a carefree life and will have {N} dollars left.**“ – where **N** is the remaining amount of money.
* If **the money is not enough**:
  + „**He will need {M} dollars to survive.**“ – where **M** is the **insufficient** amount of money.

### Sample Input and Output

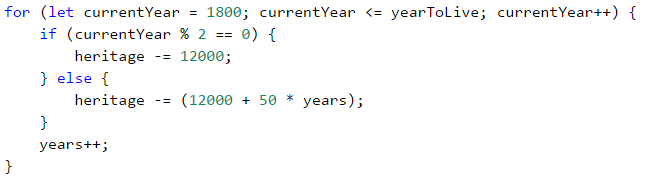
| **Input** | **Output** | **Explanations** |
| --- | --- | --- |
| 50000 1802 | Yes! Ivan would live a carefree life and would have 13050.00 USD left. | 1800 → **even**  → **Spend 12000** USD  → Remaining 50000 – 12000 = **38000** 1801 → **odd**  → **Spend** 12000 + **19\*50** = 12950 USD  → **Remaining** 38000 – 12950 = **25050** 1802 → **even**  → **Spend** 12000 USD  → **Remaining** 25050 – 12000 = **13050** |
| 100000.15 1808 | He would need 12399.85 USD to survive. | 1800 → **even**  → Remaining 100000.15 – 12000 = **88000.15** 1801 → **odd**  → **Remaining** 88000.15 – 12950 = **75050.15** **…** 1808 → **even** → -399.85 - 12000 = -12399.85 **12399.85 shortage** |

### Hints and Guidelines

The method for solving this problem is similar to the previous ones, so we should start with **declaring and initializing** the required variables. According to the problem description, Ivan is eighteen years old, so we could declare the variable **years** and set an initial value of **18**. The values of the other variables we can read from the function parameters:



Using a **for loop** we can iterate through all the years. **Starting from 1800** – the year, in which Ivan wants to return, and reach **the year until which he has to live in the past**. In the loop, we check, if the current year is **even** or **odd**. We perform the check using **division with a remainder** (**%**) by 2. If the year is **even**, we subtract from the (**heritage**) **12000**, and if the year is **odd**, we subtract from the (**heritage**) **12000 + 50 \* (the age he would have reached in the given year)**:



Finally, we have to print out the result by **checking whether the inheritance** (**heritage**) was enough to live without working or not. If the inheritance (**heritage**) is a **positive number**, we print out: "**Yes! He will live a carefree life and will have {N} dollars left.**", and if it's a **negative number**: "**He will need {M} dollars to survive.**". Don't forget to format the result with precision to the second digit after the decimal point.

**Hint**: Consider using the **Math.abs(…)** method, when printing the output in case the inheritance is insufficient.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#2>.

## Problem: Hospital

For a certain period, every day patients arrive at the hospital for examination. **Initially**, the hospital had **7 doctors**. Each doctor could treat **only one patient per day**, but sometimes there is a shortage of doctors, so **the remaining patients are sent to other hospitals**. **Every third day** the hospital makes evaluations and **if the count of untreated patients is greater than the count of treated ones, another doctor is appointed**. Each new doctor is appointed at the beginning of the day before the patients' admission.

Write a program, that calculates **for a given period, the count of treated and untreated patients**.

### Input Data

On the first line (argument) there is an integer in the range of [**1 … 1000**] - **The period**, for which you need to make calculations. On the next **lines (arguments)** there are **integers** in the range of [**1 … 10 000**] – the **number of arriving patients**, for the **current day**.

### Output Data

**Print** on the console **2 lines**:

* On the **first line**: "**Treated patients: {count of treated patients}.**"
* On the **second line**: "**Untreated patients: {count of untreated patients}.**"

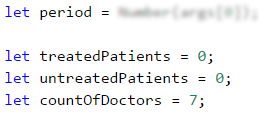
### Sample Input and Output

| **Input** | **Output** | **Explanation** |
| --- | --- | --- |
| 4 7 27 9 1 | Treated patients: 23. Untreated patients: 21. | **Day 1**: 7 treated and 0 untreated patients for the day **Day 2**: 7 treated and 20 untreated patients for the day **Day 3**: 14 patients have been treated so far and 20 untreated –> A new doctor is appointed –> 8 treated and 1 untreated patients for the day **Day 4**: 1 treated and 0 untreated patients for the day **Total: 23 treated and 21 untreated patients.** |

| **Input** | **Output** |
| --- | --- |
| 6 25 25 25 25 25 2 | Treated patients: 40. Untreated patients: 87. |
| 3 7 7 7 | Treated patients: 21. Untreated patients: 0. |

### Hints and Guidelines

We start again by, **declaring and initializing** the required variables. The calculation period we read from the console and save in the **period** variable. We will need some additional variables such as the number of treated patients (**treatedPatients**), the number of untreated patients (**untreatedPatients**), and the number of doctors (**countOfDoctors**), which initially is set to 7.



Using **for loop** we iterate through all days in the given period (**period**). For each day, we read from the console the number of the patients(**currentPatients**). According to the problem description, the increase of the doctors can occur **every third day**, **but** only if the count of untreated patients is **greater** than the count of treated ones. For this purpose we check, if the day is the third one – using the division with a remainder operator (**%**): **day % 3 == 0**.

For example:

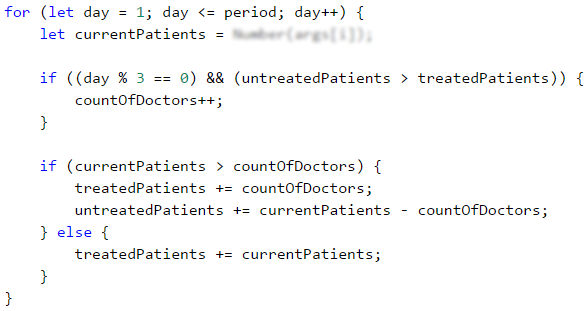
* If it's the **third** day, the remainder of the division by **3** will be **0** (**3 % 3 = 0**) and the check **day % 3 == 0** will return **true**.
* If it's the **second** day, the remainder of the division by **3** will be **2** (**2 % 3 = 2**) and the check will return **false**.
* If it's the **fourth** day, the remainder of the division by **1** (**4 % 3 = 1**) and the check will return **false**.

If **day % 3 == 0** returns **true**, the program logic will check, if the number of untreated patients is greater than the treated ones: **untreatedPatients > treatedPatients**. If the result is again **true**, then the count of doctors will be increased (**countOfDoctors**).

Then we have to check if the daily count of patients (**currentPatients**) is greater than the count of the doctors (**countOfDoctors**). If the count of patients is **greater**:

* Increase the value of the variable **treatedPatients** by the count of doctors (**countOfDoctors**).
* Increase the value of the variable **untreatdPatients** by the count of remaining patients, which we calculate by subtracting the count of doctors from the count of patients (**currentPatients - countOfDoctors**).

If the count of patients **is not greater**, we increase only the value of the variable **treatedPatients** by the count of patients for the day (**currentPatients**).



Finally, the only thing left is to print the number of treated and untreated patients.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#3>.

## Problem: Division

**N integers** are given in the range of [**1 … 1000**]. **Some percentage p1 of them are divisible without remainder by 2**, **percentage p2** are **divisible without remainder by 3**, **percentage p3** is **divisible without remainder by 4**. Write a program that calculates and prints the percentages p1, p2, and p3. **Example:** We are given **n = 10** numbers: 680, 2, 600, 200, 800, 799, 199, 46, 128, 65 and the following distribution and visualization:

| **Division without remainder by:** | **Numbers** | **Count** | **Percent** |
| --- | --- | --- | --- |
| 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%** |

### Input Data

On the first line (argument) of the input, we have integer **n** (1 ≤ **n** ≤ 1000) – count of numbers. On each next **n line**, we have **one integer** in the range of [**1 … 1000**] – numbers that have to be checked for division without remainder.

### Output Data

Print on the console **3 lines**, each containing a percentage between 0% and 100%, formatted with precision two digits after the decimal point, for example 25.00%, 66.67%, 57.14%.

* On the **first line** – the percentage of the numbers, which is **divisible by 2**.
* On the **second line** – the percentage of the numbers, which is **divisible by 3**.
* On the **third line** – the percentage of the numbers, which is **divisible by 4**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| **10** 680 2 600 200 800 799 199 46 128 65 | 70.00% 10.00% 50.00% | **3** 3 6 9 | 33.33% 100.00% 0.00% | **1** 12 | 100.00% 100.00% 100.00% |

### Hints and Guidelines

For the current and for the next problem you will have to write the program code by yourself, following the given guidelines.

The program logic that solves the current problem is similar to the **Histogram** problem, that we reviewed above. Therefore we can start with declaring the required variables. Typical variable names could be **n** – count of numbers (that we need to read from the console) and **divisibleBy2**, **divisibleBy3**, **divisibleBy4** – additional variables, containing a count of the numbers in the corresponding group.

To read and allocate each number to its corresponding group we have to iterate **for loop** from **0** to **n** (count of numbers). Each iteration of the loop should read and allocate **one single number**. The difference here is that **one number could be a part of several groups simultaneously**, therefore we have to perform **three different if checks for each number** - respectively, whether the number is divisible by 2,3, and 4 and then to increase the value of the variable that keeps the count of numbers in the corresponding group.

**Note**: **if-else** construction wouldn't work in this case, because once the condition is true, the code wouldn't perform any further checkings.

Finally, you need to print the obtained results, by following the specified format.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#4>.

## Problem: Logistics

In this problem, you are responsible for the logistics of various types of cargo **Depending on the weight** of each cargo you need a **different vehicle** and this will cost **different prices per ton**:

* Up to **3 tons** – **microbus** (200 USD per ton).
* From **over 3 to 11 tons** – **truck** (175 USD per ton).
* **Over 11 tons – train** (120 per ton).

Your task is to calculate **the average price per ton of cargo**, as well as **the percentage of the cargo** transported **by each vehicle**.

### Input Data

The program receives a **sequence of numbers** (arguments):

* On the first line (argument): **the count of cargos** that have to be transported – **integer** in the range of [**1 … 1000**].
* On the next lines, we pass the **weight of the cargo** – **integer** in the range of [**1 … 1000**].

### Output Data

Print on the console **4 lines**, as follow:

* **Line #1** – **average price per ton for a transported cargo** (formatted with precision two digits after the decimal point).
* **Line #2** – a **percentage** of the cargo, transported by **microbus** (between 0.00% and 100.00%, formatted with precision two digits after the decimal point).
* **Line #3** – a **percentage** of the cargo, transported by **truck** (between 0.00% and 100.00%).
* **Line #4** – a **percentage** of the cargo, transported by **train** (between 0.00% and 100.00%).

### Sample Input and Output

| **Input** | **Output** | **Explanation** |
| --- | --- | --- |
| 4 1 5 16 3 | 143.80 16.00% 20.00% 64.00% | The **microbus** is transporting two cargos **1** + **3**, a total of **4** tons. The **truck** is transporting one cargo: **5** tons. The **train** is transporting one cargo: **16** tons. **The sum** of all cargos is: 1 + 5 + 16 + 3 = **25** tons. Percentage of the cargo transported by **microbus** is: 4/25\*100 = **16.00%** Percentage of the cargo transported by **truck** is: 5/25\*100 = **20.00%** Percentage of the cargo transported by **train** is: 16/25\*100 = **64.00%** **The average price** per ton trasported cargo should be: (4 \* 200 + 5 \* 175 + 16 \* 120) / 25 = **143.80** |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 2 10 20 1 7 | 149.38 7.50% 42.50% 50.00% | 4 53 7 56 999 | 120.35 0.00% 0.63% 99.37% |

### Hints and Guidelines

First, we will read **the weight of each cargo** and **sum** the amount of cargo transported by **microbus**, **truck** and **train**, then we have to calculate the **total tons** of transported cargo. The next step is to calculate **the prices of each transport type** related to the transported tons and the **total price**. Finally, we will calculate and print **the total average price per ton** and **what part of the cargo as a percentage is transported by each type of vehicle**.

We declare the necessary variables such as **countOfLoads** – count of the cargos for transportation (we read them from the console), **sumOfTons** – the weight amount of all cargos, **microbusTons**, **truckTons**, **trainTons** – variables containing the amount of weight of transported cargo, respectively by the microbus, the truck, and the train.

We still need a**for loop** from **0** to **countOfLoads - 1**, to iterate through all cargo types. For each cargo, **we read weight** (in tons) and save the value in a variable, such as **tons**. Next, we add the weight of the current cargo (**tons**) to the sum of all cargo weights (**sumOfTons**). Once we have read the weight of the current cargo, **we need to determine which vehicle type will be used** (microbus, truck or train). For this purpose we should use **if-else** check:

* If the value of the variable **tons** is **less than 3**, increase the value of **microbusTons** by the value of **tons**:
* microbusTons += tons;
* In case the value of **tons** is **less than 11** - increase the value of **truckTons** by the value of **tons**.
* If the value of **tons** is **more than 11**, increase the value of **trainTons** by the value of **tons**.

Before printing the output, we have to **calculate the percentage of tons, transported by each vehicle** and the **average price per ton**. For the average price per ton, we will declare one more variable **totalPrice**, which will **sum the total price of all transported cargo** (by microbus, truck and train). We can calculate the average price by, dividing **totalPrice** by **sumOfTons**. Finally, you have to **calculate by yourself** the percentages of tons, transported by each vehicle, and print the results, keeping the format specified in the description.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/934#5>.

# Chapter 6.1. Nested Loops

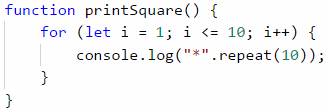
In the current chapter, we will be looking at **nested loops** and how to use for loops to **draw** various **figures on the console**, that contain symbols and signs, ordered in rows and columns on the console. We will use **single**, and **nested loops** (loops that stay in other loops), **calculations** and **checks**, to print on the console simple and not so simple figures by specified sizes.

### Problem: Rectangle of 10 x 10 Stars

Print on the console a rectangle made out of **10 x 10** stars.

| **Input** | **Output** |
| --- | --- |
| (None) | \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* |

#### Hints and Guidelines



How does the example work? We initialize **a loop with a variable i = 1**, which increases with each iteration of the loop, while it is **less or equal to 10**. This way the code in the body of the loop is executed **10 times**. In the body of the loop, we print a new line on the console **"\*".repeat(10)**, which creates a string of 10 stars.

#### Testing in The Judge System

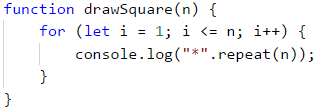
Test your solution here:: <https://judge.softuni.org/Contests/Practice/Index/935#0>.

### Example: Rectangle Made of N x N Stars

Write a program that gets a positive integer **n** and prints on the console **a rectangle made out of N x N stars**.

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 2 | \*\* \*\* | 3 | \*\*\* \*\*\* \*\*\* | 4 | \*\*\*\* \*\*\*\* \*\*\*\* \*\*\*\* |

#### Hints and Guidelines



#### Note

On some web browsers, the repeating results on the console merge in one. It's advised to use **NodeJS** for our current examples. If you still get to this case you can use the symbol **\n** at the end of the printing method **console.log("\*".repeat(10) + "\n");**.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#1>.

## Nested Loops

A **nested loop** is a construction where **the body of one loop** (the outer one) **stays inside another loop** (the inner one). In each iteration of the outer loop, **the whole** inner loop is executed. This happens in the following way:

* When nested loops start executing, **the outer loop starts** first: the controlling variable is **initialized** and after a check for ending the loop the code in its body is executed.
* After that, **the inner loop is executed**. The controlling variables' start position is initialized, a check for ending the loop is made and the code in its body is executed.
* When reaching the specified value for **ending the loop**, the program goes back one step up and continues executing the previous outer loop. The controlling variable of the outer loop changes with one step, a check is made to see if the condition for ending the loop is met and **a new execution of the nested (inner) loop is started**.
* This is repeated until the variable of the outer loop meets the condition to **end the loop**.

Here is an **example** that illustrates nested loops. The aim is again to print a rectangle made of n \* n stars, in which for each row a loop iterates from **1** to n, and for each column a nested loop is executed from **1** to \*n:

function drawSquare(n) {

for (let i = 1; i <= n; i++) {

let stars = "";

for (let j = 1; j <= n; j++) {

stars += "\*";

}

console.log(stars);

}

}

Let's look at the example above. After initializing **the first (outer) loop**, its **body**, which contains **the second (nested) loop** starts executing. By itself, it prints on one row n number of stars. After **the inner** loop **finishes** executing at the first iteration of the outer one, **the first loop will continue**, i.e. it will print an empty row on the console. **After that**, the variable of **the first** loop will be **renewed** and the whole **second** loop will be executed again. The inner loop will execute as many times as the body of the outer loop executes, in this case, n times.

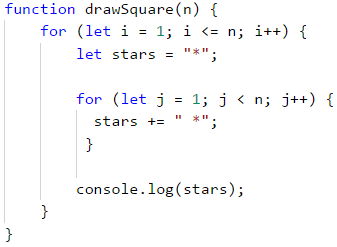
### Problem: Rectangle of N x N Stars

Print on the console a square made of **N x N** stars (use a space between the stars, staying on the same line):

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 2 | \* \* \* \* | 3 | \* \* \* \* \* \* \* \* \* | 4 | \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* |

#### Hints and Guidelines

The problem is similar to the last one. The difference here is that we need to figure out how to add a whitespace after the stars so that there aren't any excess white spaces at the beginning and the end.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#2>.

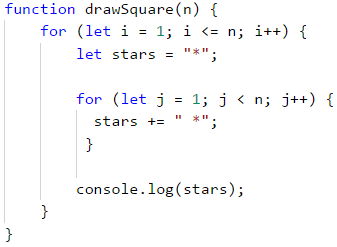
### Problem: Square of Stars

Write a program that draws on the console a square of **N x N** asterisks:

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 2 | \* \* \* \* | 3 | \* \* \* \* \* \* \* \* \* | 4 | \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* |

#### Hints and Guidelines

The problem is similar to the previous one. Here, it is necessary to consider how to print a space after the asterisks so that there are no unnecessary spaces at the beginning of the end.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#2>.

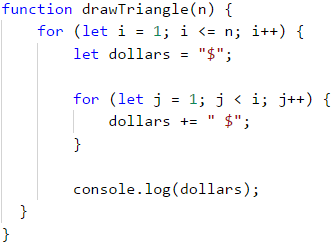
### Problem: Triangle of Dollars

Write a program that takes an integer **n** and prints **a triangle made of dollars** of size **n**.

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 3 | $ $ $ $ $ $ | 4 | $ $ $ $ $ $ $ $ $ $ | 5 | $ $ $ $ $ $ $ $ $ $ $ $ $ $ $ |

#### Hints and Guidelines

The problem is **similar** to those for drawing **a rectangle** and **square**. Once again, we will use **nested loops**, but there is **a catch** here. The difference is that **the number of columns** that we need to print depends on **the row**, on which we are and not on the input number **n**. From the example input and output data, we see that **the count of dollars depends** on which **row** we are on at the moment of the printing, i.e. 1 dollar means the first row, 3 dollars mean the third row, and so on. Let's see the following example in detail. We see that **the variable** of **the nested** loop is connected with the variable of **the outer** one. This way our program prints the desired triangle.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#3>.

### Problem: Square Frame

Write a program that takes a positive integer **n** and draws on the console **a square frame** with a size of **n \* n**.

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | + - + | - | + - + | 4 | + - - + | - - | | - - | + - - + |

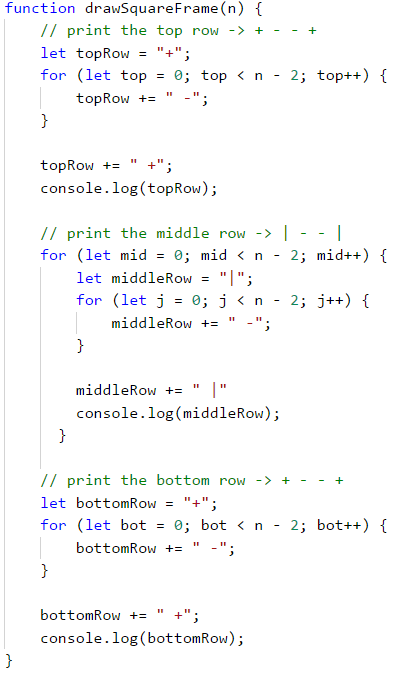
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 | + - - - + | - - - | | - - - | | - - - | + - - - + | 6 | + - - - - + | - - - - | | - - - - | | - - - - | | - - - - | + - - - - + |

#### Hints and Guidelines

We can solve the problem in the following way:

* We read from the console the number **n**.
* We print **the upper part**: first a **+** sign, then **n-2** times **-** and in the end a **+** sign.
* We print **the middle part**: we print **n-2** rows, as we first print a **|** sign, then **n-2** times **-** and in the end again a **|** sign. We can do this with nested loops.
* We print **the lower part**: first a **+** sign, then **n-2** times **-** and in the end a **+** sign.

Here is an example implementation of the above idea with nested loops:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#4>.

### Problem: Rhombus of Stars

Write a program that takes a positive integer **n** and prints **a rhombus made of stars** with size **n**.

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 | \* | 2 | \*  \* \*  \* |

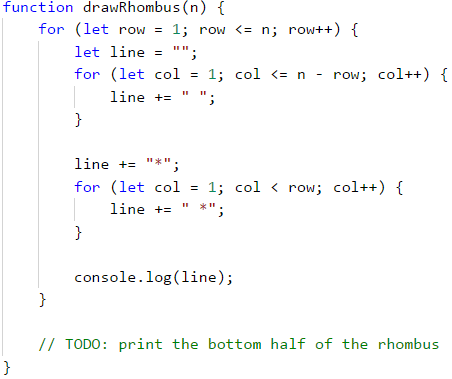
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | \*   \* \*  \* \* \*  \* \*   \* | 4 | \*   \* \*   \* \* \*  \* \* \* \*  \* \* \*   \* \*   \* |

#### Hints and Guidelines

To solve this problem, we need to mentally **divide** **the rhombus** into **two parts** – the **upper** one, which **also** includes the middle row, and the **lower** one. For **the printing** of each part, we will use **two** separate loops, as we leave the reader to decide the dependency between **n** and the variables of the loops. For the first loop we can use the following guidelines:

* We print **n-row** white spaces.
* We print **\***.
* We print **row-1** times **\***.

**The second** (lower) part will be printed **similarly**, which again we leave to the reader to do.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#5>.

### Problem: Christmas Tree

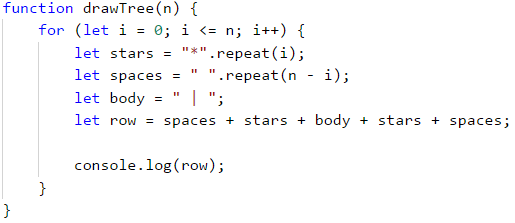
Write a program that takes a number **n** (1 ≤ n ≤ 100) and prints a Christmas tree with a height of **n+1**.

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 | |  \* | \* | 2 | |   \* | \*  \*\* | \*\* |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | |   \* | \*   \*\* | \*\*  \*\*\* | \*\*\* | 4 | |   \* | \*   \*\* | \*\*   \*\*\* | \*\*\*  \*\*\*\* | \*\*\*\* |

#### Hints and Guidelines

From the examples, we see that **the Christmas tree** can be **divided** into **three** logical parts. **The first** part is **the stars and the white spaces before and after them**, **the middle** part is **|**, and **the last** part is again **stars**, but this time there are **white spaces** only **before** them. The printing can be done with only **one loop** and the **.repeat(n)** method, which we will use once for the stars and once for the white spaces:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#6>.

## Exercises: Drawing more complex figures

Let's look at how to **draw figures** using **nested loops** with more complex logic, for which we need to think more before coding.

### Problem: Sunglasses

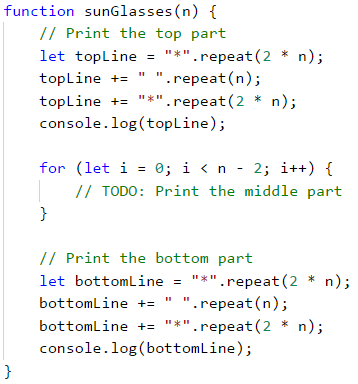
Write a program that takes an integer **n** (3 ≤ n ≤ 100) and prints sunglasses with a size of **5\*n x n** as found in the examples:

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | \*\*\*\*\*\* \*\*\*\*\*\* \*////\*|||\*////\* \*\*\*\*\*\* \*\*\*\*\*\* | 4 | \*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\* \*//////\*||||\*//////\* \*//////\* \*//////\* \*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\* |

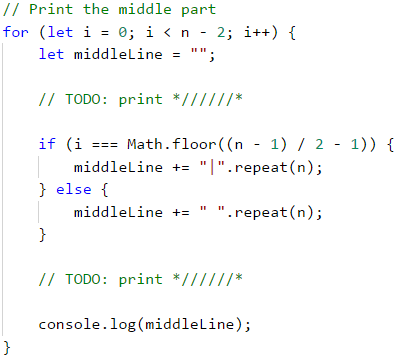
| **Input** | **Output** |
| --- | --- |
| 5 | \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* \*////////\* \*////////\* \*////////\*|||||\*////////\* \*////////\* \*////////\* \*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\* |

#### Hints and Guidelines

From the examples, we can see that the sunglasses can be divided into **three parts** – upper, middle, and lower. A part of the code with which the problem can be solved is given below. When drawing the upper and lower rows we need to print **2 \* n** stars, **n** white spaces, and **2 \* n** stars:



When drawing **the middle** part, we need to **check** if the row is **(n-1) / 2 - 1**, because in the examples we can see that in **this row** we need to print **pipes** instead of white spaces. The problem with **(n - 1) / 2 - 1** is that it can be a number with the decimal remainder. Because of this, we must use a mathematical method for removing a decimal remainder - **Math.floor(...)**. **Math.floor(...)** returns the bigger number, which is smaller or equal to the last number:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#7>.

### Problem: House

Write a program that takes a number **n** (2 ≤ **n** ≤ 100) and prints **a house** with size **n x n**, just as in the examples:

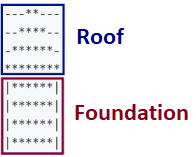
| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 2 | \*\* || | 3 | -\*- \*\*\* |\*| | 4 | -\*\*- \*\*\*\* |\*\*| |\*\*| |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 | --\*-- -\*\*\*- \*\*\*\*\* |\*\*\*| |\*\*\*| | 8 | ---\*\*--- --\*\*\*\*-- -\*\*\*\*\*\*- \*\*\*\*\*\*\*\* |\*\*\*\*\*\*| |\*\*\*\*\*\*| |\*\*\*\*\*\*| |\*\*\*\*\*\*| |

#### Hints and Guidelines

We understand from the problem explanation that the house is with a size of **n x n**. What we see from the example input and output is that:

* The house is divided into two parts: **roof and base**.



* When **n** is an even number, the point of the house is "dull".
* When **n** is odd, **the roof** is one row larger than the **base**.

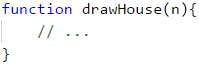
##### The Roof

* It comprises **stars** and **dashes**.
* In the top part, there are one or two stars, depending on if **n** is even or odd (also related to the dashes).
* In the lowest part, there are many stars and no dashes.
* With each lower row, **the stars** increase by 2, and **the dashes** decrease by 2.

##### The Base

* The height is **n** rows.
* It is made out of **stars** and **pipes**.
* Each row comprises 2 **pipes** – one in the beginning and one at the end of the row, and also **stars** between the pipes with a string length of **n - 2**.

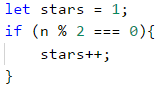
We gave **n** as a parameter to our function:



|  |  |
| --- | --- |
|  | **It is very important to check if the input data is correct!** In these tasks, it is not a problem to directly convert the data from the console into **Number**type, because it is said that we will be given valid integers. If you are making more complex programs it is a good practice to check the data. What will happen if instead of the character "A" the user inputs a number? |

To draw **the roof**, we write down how many **stars** we start with a variable called **stars**:

* If **n** is **an even** number, there will be 2 stars.
* If it is **odd**, there will be 1.



Calculate the length of **the roof**. It equals half of **n**. Write the result in the variable **roofLength**.



It is important to note that when **n** is an odd number, the length of the roof is one row more than that of the **base**. In **JavaScript** language, when you divide two numbers with a remainder, the result will be a decimal number Example:

let result = 3 / 2; // result 1.5

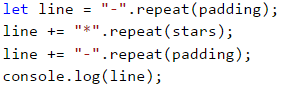
If we want to round up to the next largest integer. number, we need to use the method **Math.ceil(…)**: **let result = Math.ceil(3 / 2);**. The result from **3 / 2** is **1.5**. **Math.ceil(…)** will round the number to the next largest integer. In our case **1.5** will be rounded to **2**. **parseInt()** is used to transform the input parameter to type **Number**.

After we have calculated the length of the roof, we make a loop from 0 to **roofLength**. On each iteration we will:

* Calculate the number of **dashes** we need to draw. The number will be equal to **(n - stars) / 2**. We store it in variable **padding**.



* We print on the console: "**dash**" (**padding / 2** times) + "**stars**" (**stars** times) + "**dash**" (**padding / 2** times):

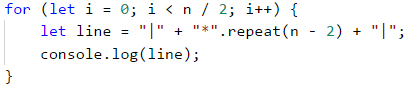


* Before the iteration is over, we add 2 to **stars** (the number of **the stars**).



After we have finished with the **roof**, it is time for **the base**. It is easier to print:

* We start with a loop from 0 to n (not inclusive).
* We print on the console: | + \* (**n - 2** times) + |.



If you have written everything as it is here, the problem should be solved.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#8>.

### Problem: Diamond

Write a program that takes an integer **n** (1 ≤ **n** ≤ 100) and prints a diamond with size **n**, as in the following examples:

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 1 | \* | 2 | \*\* | 3 | -\*- \*-\* -\*- |

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 4 | -\*\*- \*--\* -\*\*- | 5 | --\*-- -\*-\*- \*---\* -\*-\*- --\*-- | 6 | --\*\*-- -\*--\*- \*----\* -\*--\*- --\*\*-- |

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 7 | ---\*--- --\*-\*-- -\*---\*- \*-----\* -\*---\*- --\*-\*-- ---\*--- | 8 | ---\*\*--- --\*--\*-- -\*----\*- \*------\* -\*----\*- --\*--\*-- ---\*\*--- | 9 | ----\*---- ---\*-\*--- --\*---\*-- -\*-----\*- \*-------\* -\*-----\*- --\*---\*-- ---\*-\*--- ----\*---- |

#### Hints and Guidelines

What we know from the problem explanation is that the diamond is with size **n x n**.

From the example input and output we can conclude that all rows contain exactly **n** symbols, and all the rows, except for the top and bottom ones, have **2 stars**. We can mentally divide the diamond into 2 parts:

* **Upper** part. It starts from the upper tip down to the middle.
* **Lower** part. It starts from the row below the middle one and goes down to the lower tip (inclusive).

##### Upper Part

* If **n** is an **odd** number, it starts with **1 star**.
* If **n** is an **even** number, it starts with **2 stars**.
* With each row down, the stars get further away from each other.
* The space between, before, and after **the stars** is filled up with **dashes**.

##### Lower Part

* With each row down, the stars get closer to each other. This means that space (**the dashes**) between them is getting smaller and space (**the dashes**) on the left and the right is getting larger.
* The bottom-most part has 1 or 2 **stars**, depending on whether **n** is an even or odd number.

##### Upper and Lower Parts of the Diamond

* On each row, except the middle one, the stars are surrounded by inner and outer **dashes**.
* On each row, there is space between the two **stars**, except on the first and the last row (sometimes **the star is 1**).

As an entering parameter **n** of our function we give it a Number value:



We start drawing the upper part of the diamond. The first thing we need to do is to calculate the number of the outer **dashes leftRight** (the dashes on the outer side of **the stars**). It is equal to **(n - 1) / 2**, rounded down. To round the number we will use the method **Math.floor(...)** to remove the residue. We can have this case if our input is odd.



After we have calculated **leftRight**, we start drawing **the upper part** of the diamond. We can start by running a **loop** from **0** to **n / 2 + 1** (rounded down). At each iteration of the loop the following steps must be taken:

* + We draw on the console the left **dashes** (with length **leftRight**) and right after them the first **star**:



* We will calculate the distance between the two **stars**. We can do this by subtracting from **n** the number of the outer **dashes**, and the number 2 (the number of **the stars**, i.e. the diamond's outline). We need to store the result of the subtraction in a variable **mid**.



* If the **mid** is lower than 0, we know that on the row there should be only 1 star. If it is higher or equal to 0 then we have to print **dashes** with length **mid** and one **star** after them.
* We draw on the console the right outer **dashes** with length **leftRight**:



* At the end of the loop, we decrease **leftRight** by 1 (**the stars** are moving away from each other).

We are ready with the upper part.

Printing the lower part is very similar to that of the upper part. The difference is that instead of decreasing **leftRight** with 1 at the end of the loop, we will increase **leftRight** with 1 at the beginning of the loop. Also, **the loop will be from 0 to (n - 1) / 2**.

|  |  |
| --- | --- |
|  | **Repeating a code is considered bad practice** because the code becomes very hard to maintain. Let's imagine that we have a piece of code (e.g. the logic for drawing a row from the diamond) at a few more places and we decide to change it. For this, we will have to go through all the places and change it everywhere. Now let's imagine that you need to reuse a piece of code not 1, 2, or 3 times but tens of times. A way to overcome this problem is to use **functions**. You can look for additional information for functions on the Internet or look at [Chapter “10” (Functions)](https://js-book.softuni.org/chapter-10-functions.html). |

If we have written all correctly, then the problem is solved.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/935#9>.

## What Have We Learned from This Chapter?

We learned how to use the **repeat(...)** method which constructs and returns a new object from type **String**:

let foo = "\*".repeat(10);

We learned how to draw figures using nested **for** loops:

for (let i = 1; i <= n; i++) {

let stars = "";

for (let j = 1; j <= n; j++) {

stars += "\*";

}

console.log(stars);

}

## Lab: Drawing Ratings in Web

Now that we got used to **nested loops** and the way to use them to draw figures on the console, we can get into something even more interesting: we can see how loops can be used to **draw in a Web environment**. We will make a web application that visualizes a number rating (a number from 0 to 100) with stars. This kind of visualization is common in e-commerce sites, reviews of products, event rating, rating of apps, and others.

Don't worry if you don't understand all of the code, how exactly it is written and how the project works. It is normal, now we are learning to write code and we are a long way from the web development technologies. If you are struggling to write your project by following the steps, **watch the video** from the beginning of the chapter or ask for help in the SoftUni official **discussion Reddit**: <https://www.reddit.com/r/softuni/>.

### Problem: \* Ratings – Visualization in a Web Environment

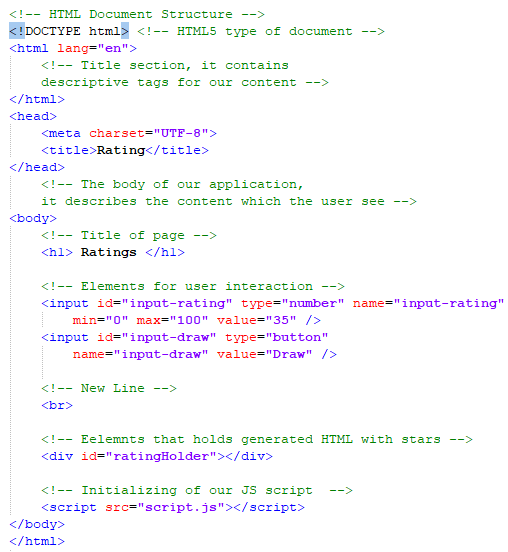
Your task is to create a **JavaScript** web application for visualizing a rating (a number from 0 to 100). From 1 to 10 stars should be drawn with halves. The starts should be generated with a for loop.

We create an empty folder in our file system with the name "**ratings**". In there we create 2 files and one folder:

* **index.html**
* **script.js**
* **images** (folder)

Now we add **images with stars** (they are part of the resources of this lab which can be downloaded from [here](https://github.com/SoftUni/Programming-Basics-Book-JS-EN/tree/master/assets/chapter-6-1-assets)). We copy them from windows explorer and paste them into the folder **images** using copy/paste.

We open **index.html** and add the following code:



This code creates one input field **input-rating**, in which the user can add a number from [**0 … 100**] and a button [**Draw**] which when clicked calculates the value of the stars with their input value. The action which will approve the input data is called **drawRating**. After that the form will print the content of **<div id="ratingHolder"></div>**. The code which is inside it will be dynamically generated HTML with a series of stars.

We add the function **drawRating()** inside the file **script.js**, which has the following code:

/\*\*

\* drawRating, draws HTML, which is needed for the visualization of the stars

\* @param {Number} rating

\* @return {String} html

\*/

function drawRating(rating) {

// string from HTML

let html = "";

// total number of stars

let allStars = 10;

// all field stars

let fullStars = Math.floor(rating / allStars);

// all empty stars

let emptyStars = Math.floor((100 - rating) / allStars);

// all half-filled stars

let halfStars = allStars - fullStars - emptyStars;

// build of HTML

for (let i = 0; i < fullStars; i++) {

html += '<img src="images/full-star.png">';

}

for (let i = 0; i < halfStars; i++) {

html += '<img src="images/half-star.png">';

}

for (let i = 0; i < emptyStars; i++) {

html += '<img src="images/empty-star.png">';

}

// return of created HTML

return html;

}

The code above takes the number **rating**, makes some calculations to find the number of **empty stars** and the number of **half-full stars**, after which it generates an HTML code, which orders a few pictures of stars one after the other so that it can make the rating picture from them. The ready HTML code is returned as a result of the function and it is ready for further use. Up to this moment, the result from this point can't be used because we can't connect it to the button. We create a function named **drawHandler()** with the following code:

/\*\*

\* drawHandler, a function that runs, when the user clicks on the button Draw

\* @return {Void}

\*/

function drawHandler() {

// Finds the input element, which stores the raiting and get its value

let ratingInput = document.getElementById("input-rating");

// By default all values from the form come as "string"

// This is why we need to convert them using "parseInt()"

let rating = parseInt(ratingInput.value);

// We find the element which holds the stars

let ratingHolder = document.getElementById("ratingHolder");

// Generated HTML from our input rating

let html = drawRating(rating);

// draw of a page

ratingHolder.innerHTML = html;

}

Our function **drawHandler()** makes several things:

* **Finds the HTML element**, which holds the rating (**input-rating**) and **gets** its value.
* **Converts value** from string to a number.
* **Finds the HTML element** that will hold the stars (**ratingHolder**).
* **Generates HTML of the stars**, using **drawRating(...)** function.
* **Places the generated HTML** using **innerHTML** method inside the element **ratingHolder**.

We need one more function that will combine the above 2 functions and connect them to the HTML elements. This function is called **appInit()**. The name hints that its role will be to initialize our application. We add the following code in our function **appInit()**:

/\*\*

\* appInit is responsible for our initial run of the application

\* @return {Void}

\*/

function appInit() {

// finds the button element inside our HTML

let button = document.getElementById("input-draw");

// Adds "click" event to perform the drawing

button.addEventListener("click", drawHandler);

// Initial draw of the rating

drawHandler();

}

After we have all the needed functions it's time to start our application. Keep in mind that **script.js** is added at the end of our file (after the closing tag of **</body>**). This is a **good practice** and gives us faster loading of the **DOM** tree. This allows us to run the following JavaScript code, which uses HTML elements. Because of these conditions, we can be sure that all conditions are already loaded inside the memory of our browser.

However, instead of calling directly **appInit()** at the end of the file, we will use one more **good practice**:

/\*\*

\* Stars the application asynchronous using "event listener".

\* Listens for "DOMContentLoaded".

\*/

document.addEventListener("DOMContentLoaded", appInit);

The event **DOMContentLoaded** confirms that the browser has finished will all actions connected to the creation of the **DOM** tree. Adding to it using **addEventListener(...)** provides it with a correct run of our JavaScript program.

When the browser is ready, it will run our starting function **appInit()**. The result from our function is:

* Hooks our function **drawHandler()** to the **click** event of the **Draw** button.
* Initial call of **drawHandler()** is to fill the stars from our current HTML.

If you have a problem with the project above **watch the video** at the beginning of this chapter. Inside the video, the application is made live step by step with a lot of explanations. You can also ask in the **Softuni Reddit**: <https://www.reddit.com/r/softuni/>.

# Chapter 6.2. Nested Loops – Exam Problems

In the previous chapter, we introduced **nested loops** and how to use them for **drawing** various kinds of **figures on the console**. We've learned how to print figures with different sizes, establishing suitable logic construction by using **single and nested for** loops in combination with different calculations and program logic:

let result = "";

for (let i = 0; i < 10; i++) {

for (let j = 0; j < 10; j++) {

result += "\*";

}

console.log(result);

result = "";

}

We also learned the **method str.repeat(count)**, which lets you for defined by us **number** of times, a **given string** to be printed:

'abc'.repeat(2); // 'abcabc'

## Exam Problems

Now let's solve some exam problems to consolidate what we have learned and to develop our algorithmic thinking.

## Problem: Draw Fort

Write a program, that reads from the console an **integer n** and draws a **fortress** with a width of **2 \* n columns** and height of **n rows**, as in the below-given examples. The left and the right inner columns have a width of **n / 2**.

### Input Data

The program input consists one element (argument) - **integer n** within the range [**3 … 1000**].

### Output Data

Print on the console **n** text lines, depicting the **fortress**, just as in the examples below.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | /^\/^\ | | \\_/\\_/ | 4 | /^^\/^^\ | | | | \\_\_/\\_\_/ |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 | /^^\\_\_/^^\ | | | | | \_\_ | \\_\_/ \\_\_/ | 8 | /^^^^\\_\_\_\_/^^^^\ | | | | | | | | | | | \_\_\_\_ | \\_\_\_\_/ \\_\_\_\_/ |

### Hints and Guidelines

By the set task condition, we can see that the **input data** will contain only one **integer** within the range [**3 … 1000**], therefore we have to create a function, that receives as an argument an **array with one element**. Because we have to work with numbers, we can use the method **Number()** as a function, to convert the received argument from a **string** to a number type:



After we've declared and initialized the input data, we have to divide the **fortress** into three parts:

* roof
* body
* base

We can see from the examples, that the **roof** consists of **two towers** and **a middle part**. Each tower has a beginning **/**, middle part **^** and an end **\**.

By the set task condition the left and the right inner columns have a width of **n / 2**, therefore we can save this value as a separate **variable**, keeping in mind, that if we receive an **odd number** as input, the result of dividing by two will be a number with a whole and fractional part. In this case, we need **only the whole part** (in the set task condition we can see, that when the input is equal to **3** the count of **^** in the inner part column is equal to **1**, and input of **5** it is **3**), we can separate it with the method **Math.trunc()** and to save only its value in our new variable:



|  |  |
| --- | --- |
|  | It's always a good practice, whenever we have an expression with the value we intend to use it **more than once**, to keep it in a variable. In this way, on the one hand, our code will be **easier to read**, and on the other hand, it will be **easier to correct** possible **errors**, as we will not have to look for each use of the expression separately. |

We also declare a second **variable**, which will keep **the value** of the part **between the two towers**. By the set task condition, we know that the total width of the fortress is **n \* 2**. In addition, we have two towers with one slash for a start and one slash for an end (a total of 4 characters), and a width of **colSize**. Therefore, to get the number of characters in the middle part, we have to subtract the size of the towers from the width of the entire fortress: **2 \* n - 2 \* colSize - 4**.

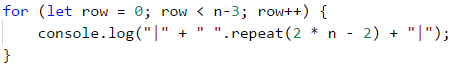


To print the **roof** part, on the console we will use the **repeat(n)** method, which joins a given string **n** number of times.

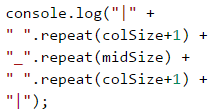


|  |  |
| --- | --- |
|  | **\** is a special symbol in JavaScript and using it solely in the method **console.log(…)**, the console will not print it out, so with **\\** we indicate on the console, that we want to print out exactly this character, without interpreting it as a special character ("**character escaping**”). |

**The fortress body** consists of beginning **|**, middle part **(white spaces)** and an end **|**. **The middle part** of white spaces has a width of **2 \* n - 2**. The number of **rows** for the walls can be determined from the given examples: **n - 3**.



To draw a penultimate row, which is part of the base, we need to print a beginning **|**, middle part **(white space)\_(white space)**, and an end **|**. To do this, we can use already declared variables **colSize** and **midSize** because as we see from the examples they are equal to the number of **\_** in the roof.



We add to the value of **white spaces** **+ 1** because in the examples we have **one** white space more.

The structure of the **fortress base** is the same as the one in the **roof**. It includes **two towers** and a **middle part**. Each **tower** begins with **\**, followed by a middle part **\_** and an end **/**.



### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/936#0>.

## Problem: Butterfly

Write a program, that takes **an integer n** from the console and draws **butterfly** with a width of **2 \* n - 1 columns** and height of **2 \* (n - 2) + 1 rows** as in the examples below. **The left and the right** **part** have a **width of n - 1**.

### Input Data

The input consists of one element (argument) - **integer n** in the range [**3 … 1000**].

### Output Data

Print on the console **2 \* (n - 2) + 1** text rows, representing the **butterfly**, exactly as shown in the examples.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | \*\ /\*  @  \*/ \\* | 5 | \*\*\*\ /\*\*\* ---\ /--- \*\*\*\ /\*\*\*  @  \*\*\*/ \\*\*\* ---/ \--- \*\*\*/ \\*\*\* |

| **Input** | **Output** |
| --- | --- |
| 7 | \*\*\*\*\*\ /\*\*\*\*\* -----\ /----- \*\*\*\*\*\ /\*\*\*\*\* -----\ /----- \*\*\*\*\*\ /\*\*\*\*\*  @  \*\*\*\*\*/ \\*\*\*\*\* -----/ \----- \*\*\*\*\*/ \\*\*\*\*\* -----/ \----- \*\*\*\*\*/ \\*\*\*\*\* |

### Hints and Guidelines

Similar to the previous task, we can see from the condition, that the **input data** will consist of only one **integer** in the range [**3 … 1000**]. We create a function, that receives an **array of one element** as an argument. Since it's a **text** (**String**) type, and we have to work with numbers, we use the **Number()** method, to convert it to a number type:



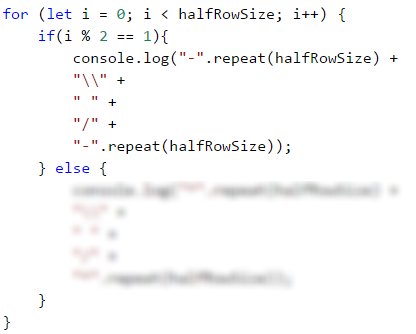
We can divide the figure into 3 parts - **upper wing**, **body**, and **lower wing**. To draw the upper wing of the butterfly, we have to divide it into three parts - a beginning with **\***, a middle part with **\ /**, and an end with **\***. After looking at the examples, we can say that the upper wing of the butterfly is with a size of**n - 2**.



To draw the upper wing we make a loop repeated **halfRowSize** number of times:



We can see in the examples, that on an **even** row we have a beginning **\***, middle part **\ /** and an end **\***, on the other hand on an **odd** row we have a beginning **-**, middle part **\ /** and an end **-**. Therefore, at each iteration of the loop, we have to do an **if-else** check to see whether the row that we print is even or odd. From the examples given in the set condition, we can see that the number of star characters and dashes on each row is equal to **n - 2**, i. e. we can use again the variable **halfRowSize** to print them.



To draw the **butterfly body**, we can use again **the variable** **halfRowSize** and print exactly **one** row on the console. The structure of the body has a beginning **(white space)**, middle part **@**, and an end **(white space)**. From the examples, we can see, that the number of the white spaces is equal to **n-1**.



What is left now is to print on the console the **lower wing**, which is **analogical to the upper wing**: we only need to swap the places of the slashes.

### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/936#1>.

## Problem: Stop

Write a program, that takes an **integer n** from the console and draws **a STOP warning sign** with size as shown in the examples below.

### Input Data

The input consists of one element (argument) - **integer n** in the range [**3 … 1000**].

### Output Data

Print on the console text lines, representing **the STOP warning sign**, just as in the examples.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | ....\_\_\_\_\_\_\_.... ...//\_\_\_\_\_\\... ..//\_\_\_\_\_\_\_\\.. .//\_\_\_\_\_\_\_\_\_\\. //\_\_\_STOP!\_\_\_\\ \\\_\_\_\_\_\_\_\_\_\_\_// .\\\_\_\_\_\_\_\_\_\_//. ..\\\_\_\_\_\_\_\_//.. | 6 | .......\_\_\_\_\_\_\_\_\_\_\_\_\_....... ......//\_\_\_\_\_\_\_\_\_\_\_\\...... .....//\_\_\_\_\_\_\_\_\_\_\_\_\_\\..... ....//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\.... ...//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\... ..//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\.. .//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\. //\_\_\_\_\_\_\_\_\_STOP!\_\_\_\_\_\_\_\_\_\\ \\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_// .\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//. ..\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//.. ...\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//... ....\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//.... .....\\\\_\_\_\_\_\_\_\_\_\_\_//..... |

| **Input** | **Output** |
| --- | --- |
| 7 | ........\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_........ .......//\_\_\_\_\_\_\_\_\_\_\_\_\_\\....... ......//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\...... .....//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\..... ....//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\.... ...//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\... ..//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\.. .//\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\\. //\_\_\_\_\_\_\_\_\_\_\_STOP!\_\_\_\_\_\_\_\_\_\_\_\\ \\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_// .\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//. ..\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//.. ...\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//... ....\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//.... .....\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//..... ......\\\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_//...... |

### Hints and Guidelines

As in the previous problem, we have to create a function that takes an array of one element, and using the **Number()** function, we convert it from text **(String)** type to number type:



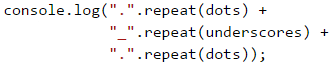
We can **divide** the figure into **three parts** - upper, middle, and lower. **The upper part** consists of two subparts - a starting line and lines in which the sign widens. **The starting line** consists of beginning **.**, middle part **\_**, and an end **.**. After looking at the examples, we can see that the beginning has a size of **n + 1** and it's better to keep this **value** as a separate **variable**.



We also have to create a second **variable**, in which we will keep **the value** of the **first row middle part** which has a size of **2 \* n + 1**.



Once we have declared and initialized the two variables, we can print the first row on the console.



To draw the rows in which the sign is getting **"wider"**, we have to create a **loop**, that iterates **n** number of times. The row structure consists of a beginning **.**, **//** + middle part **\_** + **\\** and an end **.**. To reuse the already created **variables**, we have to decrease **dots** by 1 and **underscores** by 2, because we've already **printed** the first row, and the dots and underscores in the top part of the figure are **decreasing** on each row.



At each subsequent iteration **the beginning** and **the end** decrease by 1, and **the middle part** increases by 2.

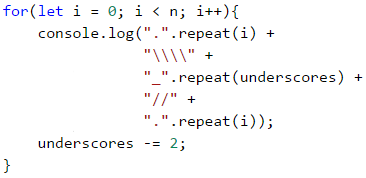


**The middle part** of the figure begins with **//** + **\_**, middle part **STOP!** and an end **\_** + **\\**. The count of the underscores **\_** is **(underscores - 5) / 2**.



**The lower part** of the figure, in which the width of the sign **decreases**, can be done by creating a **loop**, that iterates **n** number of times. The structure of a row should have a beginning **.** + **\\**, middle part **\_** and an end **//** + **.**. The number of the **dots** in the first loop iteration has to be 0 and each subsequent has to **increase** by one. Therefore we can say that the **dots in the lower part of the figure** are equal to **i**.

To ensure proper operation of our program, on each **loop** iteration, we have to **decrease** the number of **\_** by **2**.



### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/936#2>.

## Problem: Arrow

Write a program that receives from the console **an odd integer n** and draws **a vertical arrow** with size as in the examples below.

### Input Data

The input is **an odd integer n** (argument) within the range [**3 … 79**].

### Output Data

Print on the console a vertical arrow, in which "**#**" (hash sign) marks the outline of the arrow, and "**.**" - the rest.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | .###. .#.#. ##.## .#.#. ..#.. | 5 | ..#####.. ..#...#.. ..#...#.. ..#...#.. ###...### .#.....#. ..#...#.. ...#.#... ....#.... |

| **Input** | **Output** |
| --- | --- |
| 9 | ....#########.... ....#.......#.... ....#.......#.... ....#.......#.... ....#.......#.... ....#.......#.... ....#.......#.... ....#.......#.... #####.......##### .#.............#. ..#...........#.. ...#.........#... ....#.......#.... .....#.....#..... ......#...#...... .......#.#....... ........#........ |

### Hints and Guidelines

As in the previous problem, we have to create a function, that receives an array with one element and using the **Number()** method, to convert it from text to a number type:



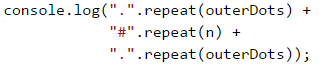
We can divide the figure into **3 parts** - upper, middle, and lower. **The upper part** consists of two subparts - first row and body of the arrow. We can see from the examples, that the count of **the outer dots** in the first row and in the body of the arrow are equal to **(n - 1) / 2**. We can keep this value in **a variable** **outerDots**:



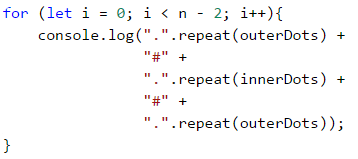
The count of **the inner dots** in the body of the arrow is equal to **(n - 2)**. We have to create **the variable** **innerDots**, which will keep this value:



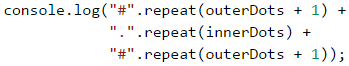
We can see from the examples the structure of the first row. We can use the declared and initialized by us **variables** **outerDots** and **n**, to print **the first row**:



To draw **the body of the arrow**, we have to create **a loop**, which iterates **n - 2** number of times:



**The middle part of the figure** is made of a beginning **#**, middle part **.** and an end **#**. The count of **#** is equal to **outerDots + 1**:

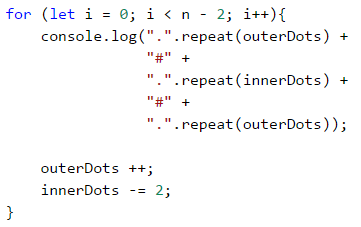


To draw **the lower part of the arrow**, we have to assign new values of the two **variables** **outerDots** and **innerDots**:

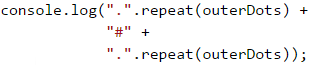


On each loop iteration **outerDots** increases by 1, and **innerDots** decreases by 2. We can notice, that on the penultimate row the **innerDots** value will be 1 and on each subsequent loop iteration will be a **negative number**. If we use **the method str.repeat(count)** with a negative number, the program will **throw an error**. To avoid that we can print the last row of the figure separately.

The height of the lower part of the arrow is **n - 1**, therefore **the loop**, that will print all the rows, except the last one, have to iterate **n - 2** number of times:



**The last row** of our figure is made of a beginning **.**, middle part **#**, and an end **.**. The count of **.** is equal to **outerDots**:



### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/936#3>.

## Problem: Axe

Write a program, that receives **an integer n** and draws an axe with size as in the example below. The width of the axe is **5 \* n** columns.

### Input Data

The input consists one element (argument) - **integer n** within range [**2..42**].

### Output Data

Print on the console **axe**, as in the examples.

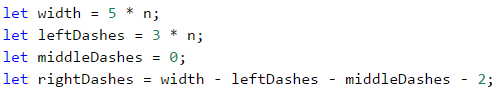
### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 | ------\*\*-- ------\*-\*- \*\*\*\*\*\*\*-\*- ------\*\*\*- | 5 | ---------------\*\*-------- ---------------\*-\*------- ---------------\*--\*------ ---------------\*---\*----- ---------------\*----\*---- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*----\*---- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*----\*---- ---------------\*----\*---- --------------\*\*\*\*\*\*\*\*--- |

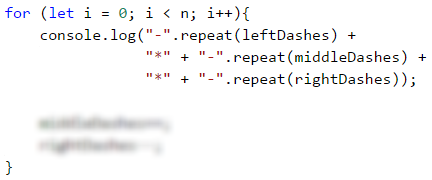
| **Input** | **Output** |
| --- | --- |
| 8 | ------------------------\*\*-------------- ------------------------\*-\*------------- ------------------------\*--\*------------ ------------------------\*---\*----------- ------------------------\*----\*---------- ------------------------\*-----\*--------- ------------------------\*------\*-------- ------------------------\*-------\*------- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------\*------- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------\*------- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------\*------- \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*-------\*------- ------------------------\*-------\*------- -----------------------\*---------\*------ ----------------------\*-----------\*----- ---------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*---- |

### Hints and Guidelines

To solve the problem, we have to calculate the **dashes in the left**, **the dashes in the middle**, **the dashes in the right**, and the whole figure length.



Once we have declared and initialized the **variables**, we can draw the figure, starting with the **upper part**. We can see from the examples what the structure of **the first row** is and we can create a loop that iterates **n** number of times. At each loop iteration, **the middle dashes** are increasing by 1, and the **right dashes** are decreasing by 1.

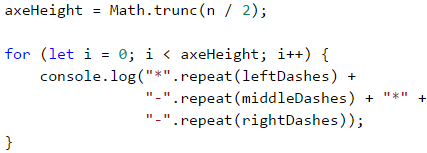


Now we have to draw **the handle of the axe**. To be able to use the newly created **variables**, when drawing the handle of the axe, we have to decrease **the middle dashes** by 1 and increase **these on the left and right** by 1.

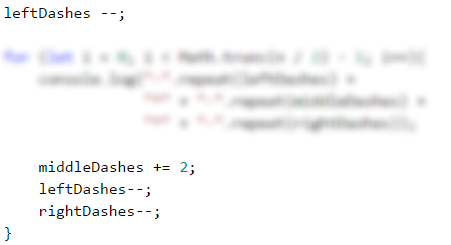


**The handle of the axe** we can draw, by iterating a loop that repeats **n / 2** number of times. We can set this value into a separate **variable**, considering that when dividing **odd number** inputs by 2 the result will be **a real number** with a whole and fractional part. Since in this case, we need **only the whole part** (from the example condition we see that at input **5** the height of the axe handle is **2**), we can use the **Math.trunc()** method, to save only its value in our new variable.

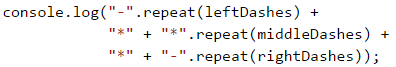
We get the structure of the handle from the examples given:



**The lower part** of the figure should be divided into two subparts - the **head of the axe** and the **last row of the figure**. We will print on the console **the head of the axe**, by making a self iterating loop **axeHeight - 1** number of times. On each iteration, **the left dashes** and **the right dashes** decrease by 1, and **the middle dashes** increase by 2.



For **the last row** of the figure, we can use again, the already declared variables **leftDashes**, **middleDashes**, **rightDashes**.



### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/936#4>.

# Chapter 7.1. Complex Loops

Since we have learned what **for** loops are and their function in code, now is the time to take a look at **other loop types**, as well as some **more complex loop constructs**. These will expand our knowledge and help us solve more challenging problems. In particular, we will discuss how to use the following programming constructs:

* loops **with step**
* **while** loops
* **do-while** loops
* **infinite** loops

In the current chapter, we will also take a look at the **break** operator and **how** to empty it to **interrupt** a loop.

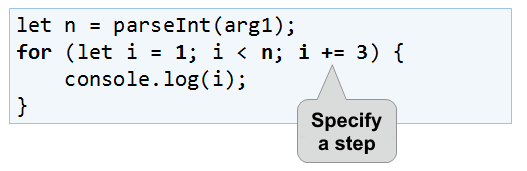
## Loops with a Step

In the **"Loops (Repetitions)"** chapter we learned how the **for** loop works and we already know when and to what purpose to use it. In the present chapter we will **take a look** at a particular and a very important **part of this structure** - its **step**.

### Loop with a Step – Explanation

The **step** is the **part** of the **for** loop, that specifies the **amount** used to **increment** or **decrement** the **main** variable. The **step** is declared in the skeleton of the **for** loop.

Most often we have a **size of 1** and in this case, instead of writing **i += 1** or **i -= 1**, we can use the shorthand operators **i++** or **i--**. In case we need the step to be **different than 1** when increasing we use the shorthand **i += + required step**, and when we decrease **i -= + required step**. With a step of 3 the loop would appear as below:



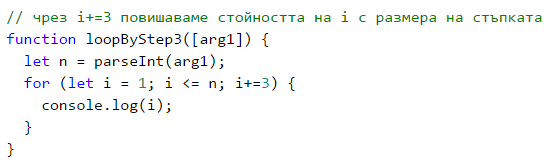
Here is a series of sample problems, the solution of which will help us better understand the use of a **step** in a **for** loop.

### Problem: Numbers 1...N with Step 3

Write a program that prints the numbers **from 1 to n** with a **step of 3**. For example, **if n = 100**, then the output would be: **1, 4, 7, 10, …, 94, 97, 100**.

We can solve the problem using the following sequence of actions (algorithm):

* We create a function that will take the number **n**.
* We run a **for loop from 1** to **n** with a step of **3**.
* in the **body of the loop**, we print the value of the current step.



#### Testing in The Judge System

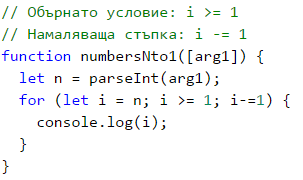
You can test your solution at the following link: <https://judge.softuni.org/Contests/Practice/Index/937#0>.

### Problem: Numbers N...1

Write a program that prints the numbers **from n to 1 in reverse** (step -1). For example, **if n = 100**, then the output will be: **100, 99, 98, …, 3, 2, 1**.

We can solve the problem in the following manner:

* We create a function that will take the number **n**.
* We run a **for loop**, by assigning **let i = n**.
* We reverse the condition of the loop: **i >= 1**.
* We define the step size: **-1**.
* in the **body of the loop**, we print the value of the current step.



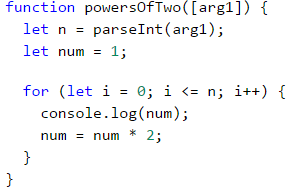
#### Testing in The Judge System

You can test your solution at the following link: <https://judge.softuni.org/Contests/Practice/Index/937#1>.

### Problem: Powers of Two

In the following example, we will use the standard size 1 step.

Write a program that prints the numbers **from 1 to 2^n** (two to the power of n). For example, **if n = 10**, then the output would be **1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024**.



#### Testing in The Judge System

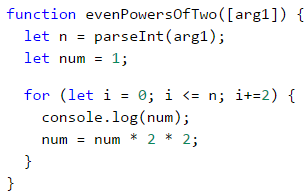
You can test your solution at the following link: <https://judge.softuni.org/Contests/Practice/Index/937#2>.

### Problem: Even Powers of 2

Print the **even** powers of **2** until **2^n**: **2^0, 2^2, 2^4, 2^8, …, 2^n**. For example, if **n = 10**, then the output would be **1, 4, 16, 64, 256, 1024**.

Here is how we can solve the problem:

* We create a function that will take the number **n**.
* We declare a variable **num** that will hold the current number and we assign it the initial **value of 1**.
* For the **step** of the loop, we set a value of **2**.
* In the **body of the loop**: we print the value of the current number and **increase the current number num 4 times** (as per the problem description).



#### Testing in The Judge System

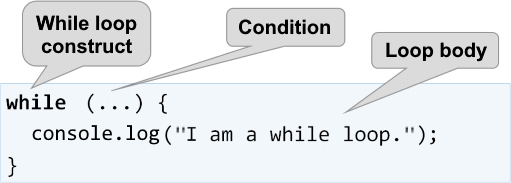
You can test your solution at the following link: <https://judge.softuni.org/Contests/Practice/Index/937#3>.

## While Loop

The next type of loops that we will familiarize are called **while loops**. The special thing about them is that they repeat a command block **while a condition is met**. Their structure is a bit different than that of the **for** loops, however, they boast a simpler syntax.

### While Loop – Explanation

In programming, the **while loop** is used when we want to **repeat** the execution of a specific logic block until **a specific condition is met**. Any expression that returns either **true** or **false** (a Boolean) can be used as a "**condition**". When the **condition** becomes **invalid**, the **while** loop **is interrupted** and the program **proceeds** to execute the code after the loop. The **while loop** structure looks like this:



Here is a series of sample problems, the solutions of which will help us better understand the use of the **while** loop.

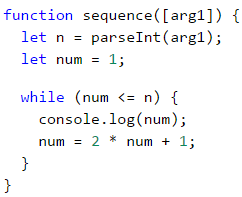
### Problem: Sequence 2k+1

Write a program that prints **all numbers ≤ n** in the series: **1, 3, 7, 15, 31, …**, assuming that each number is generated according to the following formula nextNumber = **previousNumber \* 2 + 1**.

Here is how we can solve the problem:

* We create a function that will take the number **n**.
* We declare a variable **num** that will hold the current number and we assign it the initial **value of 1**.
* For loop condition, we use **the current number <= n**.
* in the **body of the loop**: we print the value of the current number and increase the current number by using the formula above.

Here is a sample implementation of this idea:



#### Testing in The Judge System

You can test your solution here: [https://judge.softuni.org/Contests/Practice/Index/937#4](https://judge.softuni.org/Contests/Practice/Index/937#1).

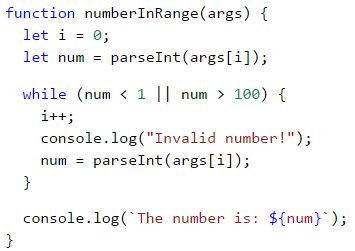
### Problem: Number in Range [1...100]

Enter an integer in the range [**1 … 100**]. If the entered number is **invalid**, enter **another**. In this case, an invalid number would be any number that is **outside** the given range.

To solve the problem, we can use the following algorithm:

* We declare variable **i**, to which we assign the initial value of **0**. Through which we save the position of each argument passed to our function.
* We declare a variable **num**, to which we assign the integer part of the first argument, passed to the function.
* For a loop condition, we put a **true** expression, if the number **is not** in the range specified in the problem description.
* On the **body of the loop**: we increment **i** so that at the next loop cycle we can take the next number passed to our function. We print the message "**Invalid number!**" on the console, afterwards we assign a new value to **num** (the next argument, passed to our function).
* Once we have validated the entered number, we print its value outside the body of the loop.

Here is a sample implementation of the algorithm using a **while loop**:



#### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#5>.

## Greatest Common Divisor (GCD)

Before proceeding to the next problem, we should become familiar with the definition of the greatest common divisor (GCD).

**Definition of GCD**: the greatest common divisor of two **natural** numbers **a** and **b** is the largest number that **divides both** **a** and **b** without reminder. For example:

| **a** | **b** | **GCD** |
| --- | --- | --- |
| 24 | 16 | 8 |
| 67 | 18 | 1 |
| 12 | 24 | 12 |
| 15 | 9 | 3 |
| 10 | 10 | 10 |
| 100 | 88 | 4 |

## The Euclidean Algorithm

In the next problem we will use one of the first published algorithms for finding the GCD – **Euclid's algorithm**:

**Until** we have a remainder of 0:

* We divide the greater number by the smaller one.
* We take the remainder of the division.

Euclid's algorithm **pseudo-code**:

while b ≠ 0

var oldB = b;

b = a % b;

a = oldB;

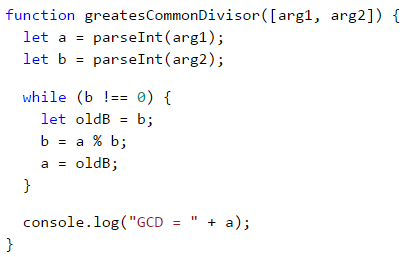
print a;

### Problem: Greatest Common Divisor (GCD)

Enter the **integers** **a** and **b** and find **GCD(a, b)**.

We will solve the problem by implementing **Euclid's algorithm**:

* We declare variables **a** and **b**, to which we assign the **integer** values, passed to our function.
* For loop condition, we use a **true** expression, if the number **b** **is different** than **0**.
* In the **body of the loop** we follow the instructions from the pseudo-code:
  + We create a temporary variable to which we assign the **current** value of **b**.
  + We assign a new value to **b**, which is the remainder of the division of **a** and **b**.
  + On variable **a** we assign the **previous** value of the variable **b**.
* Once the loop is complete and we have found the GCD, we print it on the screen.

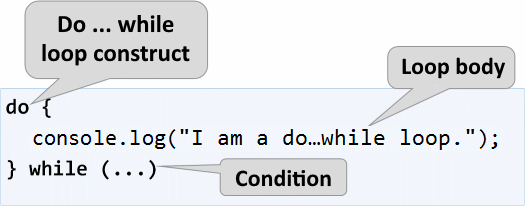


#### Testing in The Judge System

You can test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#6>.

## Do-While Loop

The next type of loop we will study is the **do-while** loop. By structure it resembles the **while**, but with a significant difference. The **do-while** will execute its body **at least once**. Why is this? In the **do-while** loop structure, the **condition** is always checked **after** the body, which ensures that the **first loop iteration** will **execute** the code and the check for **the end of the loop** will be applied to each **subsequent** iteration of the **do-while**.



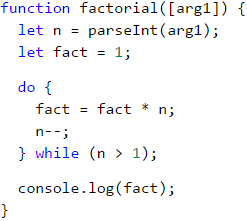
Now we should proceed with the usual set of practical problems, that will help us better understand the **do-while** loop.

### Problem: Factorial

For natural **n** number, calculate **n! = 1 \* 2 \* 3 \* … \* n**. For example, if **n = 5**, then the result would be: **5!** = 1 \* 2 \* 3 \* 4 \* 5 = **120**.

Here is how we can calculate factorial in more detail:

* We declare the variable **n**, to which we assign the integer value, passed to our function.
* We declare another variable - **fact**, with an initial value of 1. We will use it in the calculation and store the factorial value.
* For the loop condition we will use **n > 1**, because each time we perform the calculations in the body of the loop, we will decrease the value of **n** by 1.
* In the body of the loop:
  + We assign a new value to **fact**, which value is the product of multiplying the current **fact** with the current **n**.
  + We decrement **n** with **1**.
* Outside the body of the loop, we print the final factorial value.



#### Testing in The Judge System

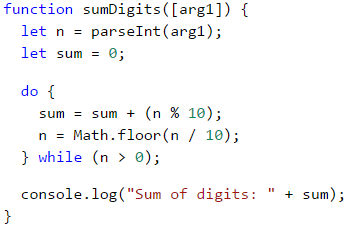
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#7>.

### Problem: Sum Digits

Sum up the digits of the integer **positive** number **n**. For example if **n = 5634**, then the output would be: 5 + 6 + 3 + 4 = **18**.

We can use the following idea to solve the problem:

* We declare the variable **n**, to which we assign a value equal to the number passed to our function.
* We create a second variable - **sum**, with an initial value of 0. We will use it for the calculation and storage of the result.
* As a loop condition, we will use **n > 0**, since, after each iteration of the loop, we will be removing the last digit from **n**.
* In the body of the loop:
  + We assign a new value to **sum**, which is the result of the sum of the current value of **sum** with the last digit of **n**.
  + We assign a new value to **n**, which is the result of removing the last digit of **n**.
* Outside the body of the loop, we print the final value of the sum.



|  |  |
| --- | --- |
|  | n % 10: **returns** the last digit of the number n. Math.floor(n / 10): **deletes** the last digit of n. |

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#8>.

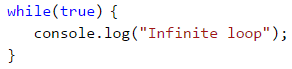
## Infinite Loops and The break Operator

So far, we were introduced to various types of loops, learning what structures they have and how they are applied. Now, we need to understand what an **infinite loop** is, when it occurs, and how we can break it using the **break** operator.

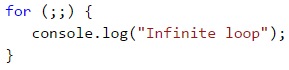
### Infinite Loop – Explanation

An infinite loop **runs infinitely** the execution of its body. With **while** and **do-while** loops the end check is a conditional expression that **always** returns **true**. Infinite **for** occurs when there is **no condition to end the loop**.

Here is an example of an **infinite while** loop:



And here is an **infinite for** loop:



### The break Operator

We already know that the infinite loop executes a certain code infinitely, but what if we want at some point under a given condition to interrupt and exit the loop? The **break** operator comes in handy in this situation.

|  |  |
| --- | --- |
|  | The operator **break** stops a loop's execution at the point it is called and the execution continues from the first line after the end of the loop. This means that the current iteration of the loop will not be completed accordingly and the rest of the code in the body of the loop will not be executed. |

### Problem: Check Prime

The next problem we are going to solve is to **check whether a given number is prime**, but before that, we should remember what are prime numbers.

**Definition**: An integer is considered **prime** if it is divisible only by itself and by 1. By definition, the prime numbers are positive and greater than 1. The smallest prime number is **2**.

We can assume that an integer **n** is a prime number if **n > 1** and **n** is not divisible by a number between **2** and **n-1**.

The first few prime numbers are: 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, …

By contrast, **composite numbers** are integers, which can be obtained by multiplying several prime numbers.

Here are some examples of composite numbers:

* **10** = 2 \* 5
* **42** = 2 3 7
* **143** = 13 \* 11

**An algorithm to check** whether a given number is **prime**: we check if **n > 1** and if **n** is divisible by **2**, **3**, …, **n-1** without remainder.

* If it is divisible by any of the numbers, it is **composite**.
* If it is not divisible by any of the numbers, then it is **prime**.

|  |  |
| --- | --- |
|  | We can optimize the algorithm by instead of checking until n-1, checking divisors only until √n. Think of the reasons why this is so. |

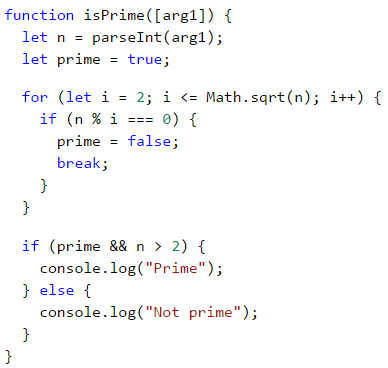
### Problem: Enter Even Number

You are tasked to write a function that takes a single input **n** integer and checks if it is prime. This can be implemented by checking if **n** is divisible by any numbers in the range between 2 and √n.

The steps of the **"prime checking algorithm"** are given below in bigger detail:

* We declare the variable **n**, to which we assign the integer passed to our function.
* We create a **prime** boolean with, and an initial value of **true**. We assume that a number is prime until proven otherwise.
* We create a **for** loop, with the initial value set to 2, for a condition the **current value <= √n**. The step is set to 1.
* In the **body of the loop**, we check if **n**, divided by the **current value** has a remainder. If there is **no reminder** from the division, then we change **prime** to **false** and exit the loop through the **break** operator.
* Depending on the value of **prime** we print whether the input number is prime (**true**) or composite (**false**).

Here is a sample implementation of the prime checking algorithm, described above:



What remains is to add a **condition that checks if the input number is greater than 1**, because, by definition numbers such as 0, 1, -1, and -2 are not prime.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#9>.

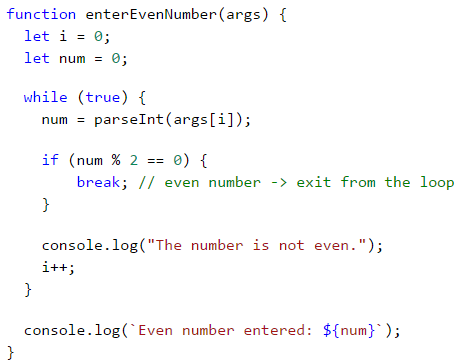
### Problem: Break Sum

Write a function, which checks whether a given number **n** is even and if so - print it on the console. An even number can be divided by 2 without a remainder. If the number is invalid, we will print that the current number is not even and the user will need to input a new number.

Here is an idea for the solution:

* We declare the variable **i**, which shall hold the initial value of **0**. Through it, we will store the position of every number passed to our function.
* We declare a variable **num**, with an initial value of **0**.
* We create an infinite **while** loop with a condition set to **true**.
* In **the body of the loop**:
  + We take the integer value, passed to our function, and assign it to **num**.
  + If **the number is even**, we exit the loop by a **break**.
  + **Otherwise**, we print a message stating that **the number is not even**. We increment **i** so that we can take the next number from the input on the next iteration. The iterations continue until an even number is entered.
* We print the even number on the console.

Here is an example implementation of the idea:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#10>.

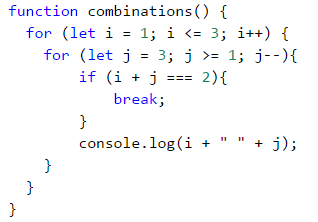
## Nested Loops and The break Operator

Now since we know what the **nested loops** are and how the **break** operator works, it is time to figure out how they work together. To get a better idea, we should write a **function** step by step, that should make all possible combinations of **number pairs**. The first number in the pair is increasing from 1 to 3, while the second one is decreasing from 3 to 1. Our solution must continue running until **i + j** **is not** equal to 2 (i.e. **i = 1** and **j = 1**).

The desired result is:



Here is a **wrong implementation**, that looks right at first glance:



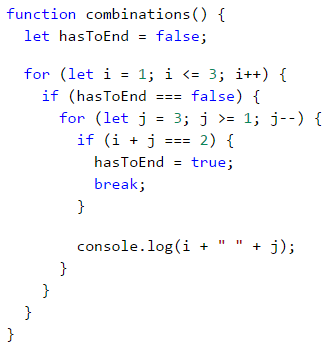
If we leave our function that way, our result will be as follows:



Why is this so? As we can see, the result is **missing "1 1"**. When the function reaches the point when **i = 1** and **j = 1**, it enters the **if** check and executes the **break** operation. This way, it **exits the inner loop**, but then continues the execution of the **outer loop**. **i** increases, the function enters the inner loop and prints the result.

|  |  |
| --- | --- |
|  | When we use the **break operator** in a **nested loop**, it interrupts **only** the execution of the inner loop. |

What is the **correct solution**? One way to solve the problem is by declaring a **bool variable**, to keep track if the loop iteration has to continue. If we have to exit (leave all nested loops), we set the variable to **true** and exit the inner loop with **break** and in the next check, we exit the outer loop. Here is an example implementation of this idea:



In this manner, when **i + j = 2**, the program will set the variable **hasToEnd = true** and will exit the inner loop. Upon the next iteration of the outer loop, via the **if** check, the function will not reach the inner loop and will interrupt its execution.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#11>.

## Exercises with loops

In this chapter, we got familiar with a few new types of loops that can perform repetitions with more complex programming logic. Let's solve a few practical problems using these new constructs.

### Problem: Fibonacci

Fibonacci's numbers in mathematics form a sequence that looks like this: **1, 1, 2, 3, 5, 8, 13, 21, 34, …**.

**The formula** to derive the next member of Fibonacci's sequence is:

F0 = 1

F1 = 1

Fn = Fn-1 + Fn-2

#### Sample Input and Output

| **Input (n)** | **Output** | **Comment** |
| --- | --- | --- |
| 10 | 89 | F(11) = F(9) + F(8) |
| 5 | 8 | F(5) = F(4) + F(3) |
| 20 | 10946 | F(20) = F(19) + F(18) |
| 0 | 1 |  |
| 1 | 1 |  |

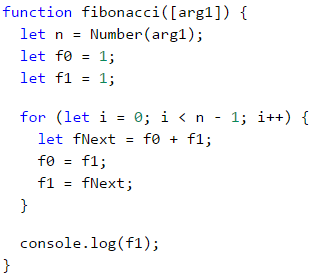
Enter an **integer** number **n** and calculate the **n-th Fibonacci number**.

#### Hints and Guidelines

An idea to solve the problem:

* We declare a **variable n**, which will hold the integer value passed to our function.
* We create the variables **f0** and **f1**, to which we assign the value to **1** since this is the start of Fibonacci's sequence.
* We create a **for** loop with condition **current value i < n - 1**.
* In the **body of the loop:**
  + We create a **temporary** variable **fNext**, to which we assign the next number in the Fibonacci sequence.
  + To **f0** we assign the current value of **f1**.
  + To **f1** we assign the value of the temporary variable **fNext**.
* Out of the loop we print the n-th number of Fibonacci.

Example implementation:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#12>.

### Problem: Number Pyramid

Print the **numbers 1 … n in a pyramid** as per the below example. On the first row, we print one number, at the second we print two, at the third, we print three, and so on, until the numbers are over. On the last line, we print as many numbers as we get until we get to **n**.

#### Sample Input and Output

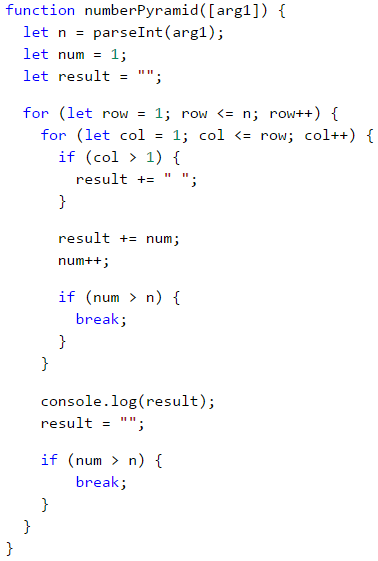
| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 7 | 1 2 3 4 5 6 7 | 5 | 1 2 3 4 5 | 10 | 1 2 3 4 5 6 7 8 9 10 |

#### Hints and Guidelines

We can solve the problem with **two nested loops** (by rows and columns) by printing in them and leaving when the last number is reached. Here is the idea, written in more detail:

* We declare a variable **n**, to which we assign the integer value passed to our function.
* We declare a variable **num** with an initial value of **1**. It will hold the count of printed numbers. At each iteration, we will **increment** it by **1** and will add it to the current row.
* We declare a variable **result**, which will hold the current row and to which we will add the value of the current cell.
* We create an **outer** **for** loop, which will be responsible for the **rows** in the table. The loop variable will be named **row** and we assign it an initial value of **1**. For condition, we set **row < n**. The step will also be **1**.
* In the body of the loop, we create an **inner** **for** loop, which will be responsible for the **columns** in the table. We name the variable **col** and assign it an initial value of **1**. For condition we set **col < row** (**row** = number of digits per line). The step will also be **1**.
* In the body of the nested loop:
  + We check whether **col > 1**, if true we add a space to the variable **result**. If we do not make this check and instead add the space every time, each resulting line will start with a space.
  + We **store** the number **num** in the current cell of the table and **increment it by 1**.
  + We check if **num > n**. If the **num** is greater than **n**, we **interrupt** the **inner loop**.
* We print the value of the variable **result** and after that, we set it to an empty value. This way we can proceed to the next line.
* Again, we check if **num > n**. If it is greater, we **interrupt** the execution of **our function** with a break.

Here is an example implementation:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#13>.

### Problem: Number Table

Print the numbers 1 … n in a table as per the examples below:

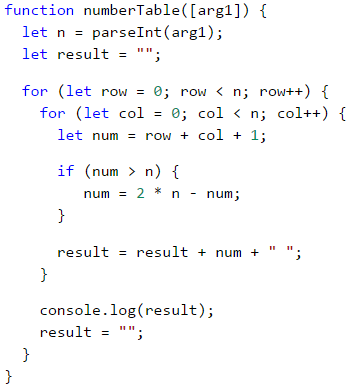
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 | 1 2 3 2 3 2 3 2 1 | 4 | 1 2 3 4 2 3 4 3 3 4 3 2 4 3 2 1 |

#### Hints and Guidelines

We can solve the problem by using **two nested loops** and some simple calculations in them:

* We take the size of the table from the integer value of the variable **n**, which is passed to our function.
* We declare the variable **result**, which will hold the current row and to which we will add the value of the current cell.
* We create a **for** loop, that will be responsible for the rows of the table. We name the loop variable **row** and assign it an initial value of **0**. For condition, we set **row < n**. The size of the step is **1**.
* In **the body of the loop** we create a nested **for** loop, that will be responsible for the columns in the table. We name the loop variable **col** and assign it an initial value of **0**. For condition, we set **col < n**. The size of the step is set to **1**.
* In **the body of the nested loop**:
  + We create a variable **num**, to which we assign the result of **the current row + the current column + 1** (+1 is needed since we count from 0).
  + We check whether **num > n**. If **num** is **greater** than **n**, we assign **num** a new value equal to **two times** n **- the current value of** num**. We do this in order not to exceed** n\*\* in any of the cells of the table.
    - We add the number of the current cell to the variable **result**.
* We print the value of the **result**, after which we set it to a new empty value. This way we can proceed to the next line.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/937#14>.

## What Have We Learned from This Chapter?

We can use **for** loops with a **step**:

for (let i = 1; i <= n; i+=3) {

console.log(i);

}

The **while** / **do-while** loops are repeated while a **condition is true**:

let num = 1;

while (num <= n) {

console.log(num++);

}

If we have to **interrupt** the loop execution, we use the **break** operator:

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: ${num}`);

# Chapter 7.2. More Complex Loops – Exam Problems

We already learned how to execute **a block of commands more than once** using a **for** loop. In the previous chapter, **we reviewed some loop structures** that would help us solve more complex problems like:

* **loops with a step**
* **nested** loops
* **while** loops
* **do-while** loops
* **infinite** loops and breaking out of the loop (**break** operator)
* **try-catch** construction

## Exam Problems

Let's start work on solving the following practice exam problems.

## Problem: Stupid Password Generator

Write a program that enters two integers **n** and **l** and generates in alphabetical order all possible **"dumb" passwords** that consist of the following **5 characters**:

* Character 1: digit from **1** to **n**.
* Character 2: digit from **1** to **n**.
* Character 3: small letter among the first **l** letters of the Latin alphabet.
* Character 4: small letter among the first **l** letters of the Latin alphabet.
* Character 5: digit from **1** to **n, bigger than first 2 digits**.

### Input Data

The input is an array with **two integers** (arguments): **n** and **l** within the range [**1 … 9**].

### Output Data

Print on the console **all "dumb" passwords in alphabetical order**, separated by **space**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| [ '2', '4' ] | 11aa2 11ab2 11ac2 11ad2 11ba2 11bb2 11bc2 11bd2 11ca2 11cb2 11cc2 11cd2 11da2 11db2 11dc2 11dd2 | [ '3' , '1' ] | 11aa2 11aa3 12aa3 21aa3 22aa3 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| [ '3' , '2' ] | 11aa2 11aa3 11ab2 11ab3 11ba2 11ba3 11bb2 11bb3 12aa3 12ab3 12ba3 12bb3 21aa3 21ab3 21ba3 21bb3 22aa3 22ab3 22ba3 22bb3 | [ '4' , '2' ] | 11aa2 11aa3 11aa4 11ab2 11ab3 11ab4 11ba2 11ba3 11ba4 11bb2 11bb3 11bb4 12aa3 12aa4 12ab3 12ab4 12ba3 12ba4 12bb3 12bb4 13aa4 13ab4 13ba4 13bb4 21aa3 21aa4 21ab3 21ab4 21ba3 21ba4 21bb3 21bb4 22aa3 22aa4 22ab3 22ab4 22ba3 22ba4 22bb3 22bb4 23aa4 23ab4 23ba4 23bb4 31aa4 31ab4 31ba4 31bb4 32aa4 32ab4 32ba4 32bb4 33aa4 33ab4 33ba4 33bb4 |

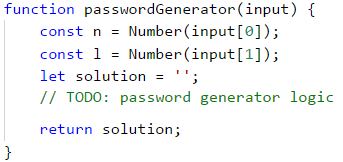
### Hints and Guidelines

We can split the solution of the problem into 3 parts:

* **Reading the input** – in the current problem, this includes reading two numbers **n** and **l**, and converting them to numbers.
* **Processing the input data** – using nested loops to iterate through every possible symbol for each of the five password symbols.
* **Printing the output** – printing every "dumb" password that meets the requirements.

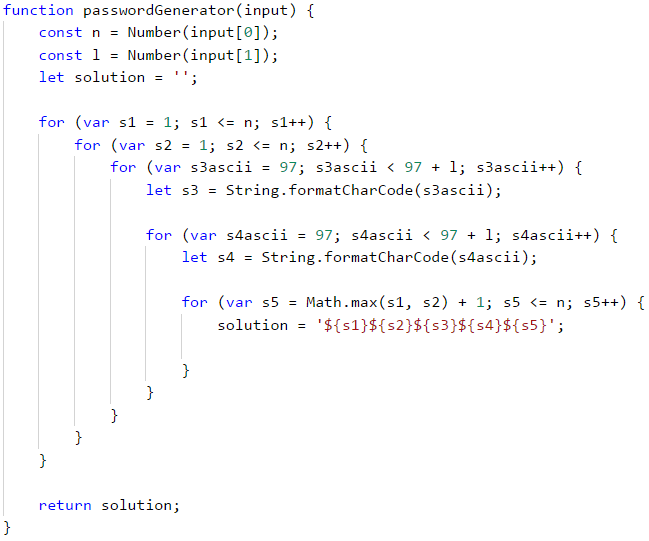
#### Reading The Input Data

For **reading** of **input** data we will declare two constants **const**: **n** and **l** and also we will declare a variable solution in which we will store a **string** with the result.



#### Printing Output

One of the ways to find a solution for this problem is to create **five** **for** nested loops, one for each variable. To ensure that the last digit is **greater** than the first two, we will use the built-in method **Math.max(…)**:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/938#0>.

## Problem: Magic Numbers

Write a program that enters a single integer **magic** number and produces all possible **6-digit numbers** for which **the product of their digits is equal to the magical number**.

Example: "Magic number" → 2

* 111112 → 1 \* 1 \* 1 \* 1 \* 1 \* 2 = 2
* 111121 → 1 \* 1 \* 1 \* 1 \* 2 \* 1 = 2
* 111211 → 1 \* 1 \* 1 \* 2 \* 1 \* 1 = 2
* 112111 → 1 \* 1 \* 2 \* 1 \* 1 \* 1 = 2
* 121111 → 1 \* 2 \* 1 \* 1 \* 1 \* 1 = 2
* 211111 → 2 \* 1 \* 1 \* 1 \* 1 \* 1 = 2

### Input Data

The program reads **one integer** (argument) within the range [**1 … 600 000**].

### Output Data

Print on the console **all magic numbers**, separated by **space**.

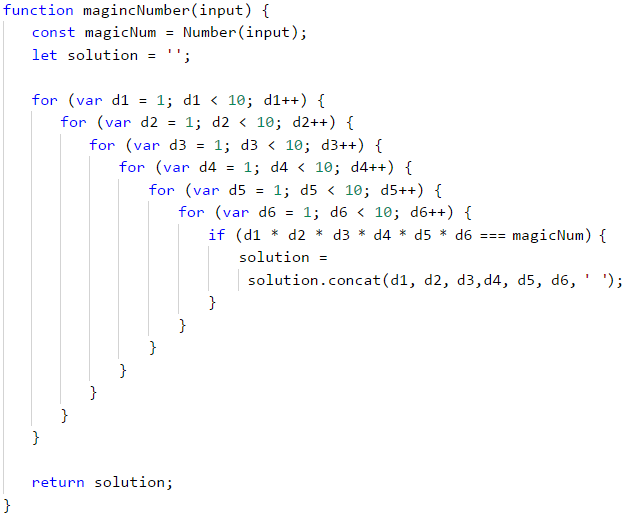
### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 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 | 531441 | 999999 |

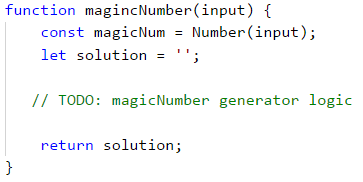
### Hints and Guidelines

**The solution** of this problem follows the **same** conception (we have to generate all n combinations of the element). Follow the steps and try to solve the problem on your own:

* Initialize the **variable** where you will store the magic number.
* Nest **six for loops**, one for each digit.
* In the last loop, using an **if** statement, check if the **product** of the six digits is **equal** to the **magic** number.

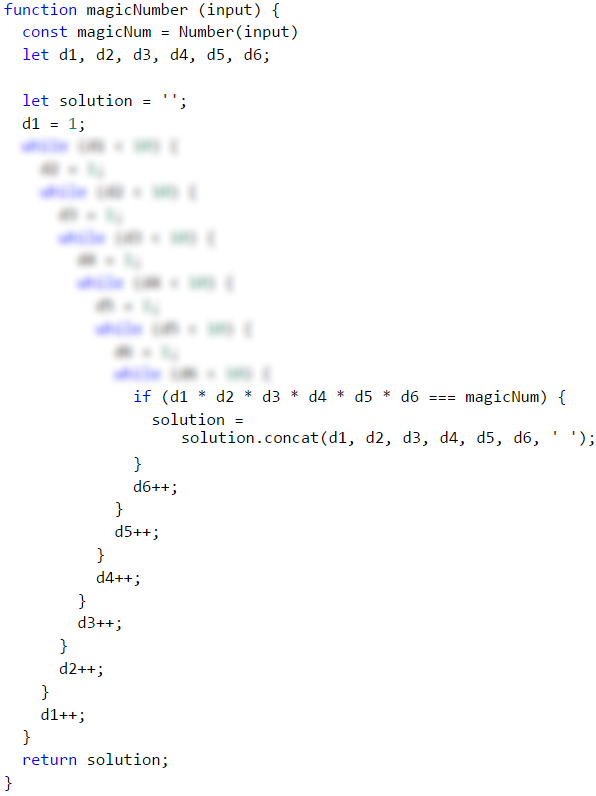


In the previous chapter, we reviewed other loop constructions. Let's look at the sample solution of the same problem using the **while** loop. First, we need to store the **input magical number** in a suitable variable. Then we will initialize 6 variables – one for each of the six digits of **the searched numbers**.



Then we will start writing **while** loops.

* We will initialize **first digit**: **d1 = 1**.
* We will set a **condition for each** loop: the digit will be less than or equal to 10.
* At the **beginning** of each loop, we set a value of the **next** digit, in this case: **d2 = 1**. In the nested **for** loops, we initialize the variables in the inner loops at each increment of the outer ones. We want to do the same here.
* At **the end** of each loop, we will increase the digit by one: **d++**.
* In the **innermost** loop, we will make **the check** and if necessary, we add the result into the variable, that stores the solution.



As we can see, we can solve a problem using different types of loops. Of course, each task has its most appropriate choice. To practice each type of loop – try to solve each of the following problems with all the learned loops.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/938#1>.

## Problem: Stop Number

Write a program that prints on the console all numbers from **N** to **M**, that are **divisible by 2** and **3 without remainder**, in **reversed order**. We will read one more "stop" number from the console – **S**. If any of the numbers divisible by 2 and 3 **is equal to the stop number, it should not be printed**, and the program should end. **Otherwise print all numbers up to N**, that meet the condition.

### Input Data

Read from the console 3 numbers, each on a single line:

* **N** – integer number: **0 ≤ N < M**.
* **M** – integer number: **N < M ≤ 10000**.
* **S** – integer number: **N ≤ S ≤ M**.

### Output Data

Print on the console on a single line all numbers, that meet the condition, separated by space.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 1 30 15 | 30 24 18 12 6 | Numbers from 30 to 1, that are divisible at the same time by 2 and 3 without remainder are: 30, 24, 18, 12 and 6. The number 15 **is not equal** to any, so the sequence **continues**. |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 1 36 12 | 36 30 24 18 | Numbers from 36 to 1, that are divisible at the same time by 2 and 3 without remainder are: 36, 30, 24, 18, 12 and 6. The number 12 **is equal** to the stop number, so **we stop by 18**. |

| **Input** | **Output** |
| --- | --- |
| 20 1000 36 | 996 990 984 978 972 966 960 954 948 942 936 930 924 918 912 906 900 894 888 882 876 870 864 858 852 846 840 834 828 822 816 810 804 798 792 786 780 774 768 762 756 750 744 738 732 726 720 714 708 702 696 690 684 678 672 666 660 654 648 642 636 630 624 618 612 606 600 594 588 582 576 570 564 558 552 546 540 534 528 522 516 510 504 498 492 486 480 474 468 462 456 450 444 438 432 426 420 414 408 402 396 390 384 378 372 366 360 354 348 342 336 330 324 318 312 306 300 294 288 282 276 270 264 258 252 246 240 234 228 222 216 210 204 198 192 186 180 174 168 162 156 150 144 138 132 126 120 114 108 102 96 90 84 78 72 66 60 54 48 42 |

### Hints and Guidelines

The problem can be divided into **four** logical parts:

* **Reading** the input.
* **Checking** all numbers in the given range, and then running a **loop**.
* **Checking** the conditions of the problem according to every number in the given range.
* **Printing** the numbers.

The **First** part is ordinary – we read **three** integer numbers from the console.

We have already seen examples of the **second** part – initialization of the **for** loop. It is a bit **tricky** – the explanation mentions that the numbers have to be printed in **reversed order**. This means that the **initial** value of the variable **i** will be **bigger**, and from the examples, we can see that it is **M**. Thus, the **final** value of **i** should be **N**. The fact that we will print the results in reversed order and the values of **i**, suggests that the step would be **decreased by 1**.



After we have initialized the **for** loop, it is time for the **third** part of the problem – **checking** the condition if the given **number is divisible both by 2 and 3 without remainder**. We will do this using one simple **if** condition that we will leave to the reader to do by themselves.

Another **tricky** part of this problem is that apart from the above check we need to do **another** one – whether the **number is equal to the "stop" number** entered from the console on the third line. To do this check, the previous one has to be passed. For this reason, we will add another **if** statement that we will **nest in the previous one**. If the condition is **true**, we need to stop the program from printing. We can do this using a **break** operator, and it will lead us **out** of the **for** loop.

If the **condition** that checks whether the number is equal with the "stop" number returns a **false** result, our program should **continue to print**. This covers the **fourth and last** part of our program.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/938#2>.

## Problem: Special Numbers

Write a program that **reads one integer number N** and generates all possible **special numbers** from **1111** to **9999**. To be considered **special**, a number must correspond to the **following condition**:

* **N to be divisible by each of its digits without remainder**.

**Example:** upon **N = 16, 2418** is a special number:

* 16 / 2 = 8 **without remainder**
* 16 / 4 = 4 **without remainder**
* 16 / 1 = 16 **without remainder**
* 16 / 8 = 2 **without remainder**

### Input Data

The input is read from the program and consists of **one integer** (argument) within the range **[1 … 600 000]**.

### Output Data

Print on the console **all special numbers**, separated by **space**.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 3 | 1111 1113 1131 1133 1311 1313 1331 1333 3111 3113 3131 3133 3311 3313 3331 3333 | 3 / 1 = 3 without remainder 3 / 3 = 1 without remainder 3 / 3 = 1 without remainder 3 / 3 = 1 without remainder |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 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 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 |

### Hints and Guidelines

Solve the problem by yourself using what you learned from the previous two problems. Keep in mind the difference between operators for **integer division ( / )** and **division with remainder ( % )** in JavaScript.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/938#3>.

## Problem: Digits

Write a program that reads from the console 1 integer within the range [**100 … 999**], and then prints it a predefined number of times – modifying it before each print, as follows:

* If the number is divisible by **5** without remainder, **subtract** from it **its first digit**.
* If the number is divisible by **3** without remainder, **subtract** from it **its second digit**
* If none of the above-mentioned conditions is valid, **add** to it **its third digit**.

Print on the console **N lines** and each line has **M numbers**, that are the result of the above actions.

* N = sum of the first and second digits of the number.
* M = sum of the first and third digits of the number.

### Input Data

The input is read from the program and is **an integer** (argument) within the range [**100 … 999**].

### Output Data

Print on the console **all integer numbers**, that is the result of the above-mentioned calculations in the respective number of rows and columns as in the examples.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 132 | 129 126 123 120 119 121 123 120 119 121 123 120 | (1 + 3) = 4 and (1 + 2) = 3 → 4 lines with 3 numbers in each Input number 132 132 → division by 3 → 132 - 3 = = 129 → division by 3 → 129 - 3 = = 126 → division by 3 → 126 - 3 = = 123 → division by 3 → 123 - 3 = = 120 → division by 5 → 120 - 1 = ..... 121 → neither by 5, nor 3 → 121 + 2 = 123 |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 376 | 382 388 394 400 397 403 409 415 412 418 424 430 427 433 439 445 442 448 454 460 457 463 469 475 472 478 484 490 487 493 499 505 502 508 514 520 517 523 529 535 532 538 544 550 547 553 559 565 562 568 574 580 577 583 589 595 592 598 604 610 607 613 619 625 622 628 634 640 637 643 649 655 652 658 664 670 667 673 679 685 682 688 694 700 697 703 709 715 712 718 | 10 lines with 9 numbers in each Input number 376 → neither 5, nor 3 → 376 + 6 → = = 382 → neither 5, nor 3 → 382 + 6 = = 388 + 6 = 394 + 6 = 400 → division by 5 → 400 - 3 = 397 |

### Hints and Guidelines

Solve the problem **by yourself**, using what you learned from the previous ones. Remember that you will need to define **different** variables for each digit of the input number.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/938#4>.

# Chapter 8.1. Practical Exam Preparation - Part I

In **the present chapter**, we will examine a few **problems** with a level of **difficulty** that can be expected in **the problems** of the practical **exam**. We will **review** and **practice** all the knowledge that was gained from this book.

## The Practical Exam

There are **6** problems included, and you will have **4 hours** to solve them. **Each** of the exam problems will **cover** one of the studied **topics** during the course. Problem topics are as follows:

* Problem with simple calculations (without conditions)
* Problem with simple condition
* Problem with more complex conditions
* Problem with a simple loop
* Problem with nested loops (drawing a figure on the console)
* Problem with nested loops and more complex logic

## The Online Evaluation System (Judge)

**All exams and homework** are automatically **tested** through the online **Judge system**: [https://judge.softuni.org](https://judge.softuni.org/). For **each** of the problems, there are **visible** (zero points) tests that help you understand what is expected of the problem and fix your mistakes, as well as **competition** tests that are **hidden** and check if your solution is working properly.

**How** does the testing in the **Judge** system work? **You upload** the source code and from the menu below you choose to compile as a **JavaScript** program. The program is being **tested** with a series of tests, giving **points** for each **successful** test.

## Problems with Simple Calculations

**The first** problem of the practical exam covers **simple calculations without checks and loops**. Here are a few examples:

### Problem: Triangle Area

|  |  |
| --- | --- |
| **Triangle in the plain** is defined by the coordinates of its three vertices. First, **the vertex (x1, y1)** is set. Then the other two vertices are set: **(x2, y2)** and **(x3, y3)** which **lie on a common horizontal line** i.e. they have the same Y coordinates). Write a program that calculates **the triangle area** by the coordinates of its three vertices. |  |

#### Input Data

We submit **6 integers** as parameters of the function: **x1, y1, x2, y2, x3, y3.**

* All input numbers are in the range [**-1000 … 1000**].
* It's guaranteed that **y2 = y3**.

### Output Data

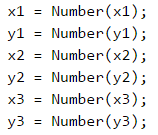
Print on the console **the triangle area**.

#### Sample Input and Output

| **Input** | **Output** | **Visualization** | **Comments** |
| --- | --- | --- | --- |
| 5 -2 6 1 1 1 | 7.5 |  | The side of the triangle **a** = 6 - 1 = **5** The height of the triangle **h** = 1 - (-2) = **3** The area of the triangle **S** = a \* h / 2 = 5 \* 3 / 2 = **7.5** |

| **Input** | **Output** | **Visualization** | **Comments** |
| --- | --- | --- | --- |
| 4 1 -1 -3 3 -3 | 8 |  | The side of the triangle **a** = 3 - (-1) = **4** The height of the triangle **h** = 1 - (-3) = **4** The area of the triangle **S** = a \* h / 2 = 4 \* 4 / 2 = **8** |

#### Hints and Guidelines

It is important in such types of tasks where some coordinates are given to pay attention to **the order** in which they are submitted and to properly understand which of the coordinates we will use and how. In this case, the input is in order **x1, y1, x2, y2, x3, y3**. If we do not follow this sequence, the solution becomes wrong. First, we write the code that reads the input data: 

We have to calculate **the side** and **the height** of the triangle. From the examples and the condition **y2 = y3** we notice that one **side** is always parallel to the horizontal axis. It means that its **length** is equal to the length of the segment between its coordinates **x2 and x3**, which is equal to the difference between the larger and the smaller coordinates. Similarly, we can calculate **the height**. It will always be equal to the difference between **y1 and y2**(or **y3**, as they are equal). Since we do not know if **x2** is greater than **x3**, or **y1** will be below or above the triangle side, we will use **the absolute values** of the difference to always get positive numbers because one segment cannot have a negative length.



We will calculate the triangle area by using our familiar formula for finding an **area of a triangle**.



The only thing left is to print the area on the console.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#0>.

### Problem: Bricks

Construction workers have to transfer a total of **x bricks**. **The workers** are **w** and work simultaneously. They transport the bricks in trolleys, each with a **capacity of m** bricks. Write a program that reads the integers **x**, **w**, and **m**, and calculates **what is the minimum number of courses** the workers need to do to transport the bricks.

#### Input Data

As parameters of the function we give **3 integers**:

* **The number of bricks x**
* **The number of workers w**
* **The capacity of the trolley m**

All input numbers are integers in the range [**1 … 1000**].

### Output Data

Print on the console **the minimum number of courses** needed to transport the bricks.

#### Sample Input and Output

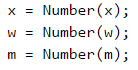
| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 120 2 30 | 2 | We have **2** workers, each transporting **30** bricks per course. In total, workers are transporting **60** bricks per course. To transport **120** bricks, exactly **2** courses are needed. |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 355 3 10 | 12 | We have **3** workers, each transporting **10** bricks per course. In total, workers are transporting **30** bricks per course. To transport **355** bricks, exactly **12** courses are needed: **11** complete courses carry **330** bricks and the last **12th** course carries the last **25** bricks. |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 5 12 30 | 1 | We have **5** workers, each transporting **30** bricks per course. In total, workers are transporting **150** bricks per course. To transport **5** bricks, only **1** course is enough (although incomplete, with only 5 bricks). |

#### Hints and Guidelines

The input is standard, and we only need to be careful about the sequence in which we read the data.



We calculate how many **bricks** the workers transport in a single course.



By dividing the total number of **bricks transported for 1 course**, we will obtain the number of **courses** needed to carry them. We will use the method **Math.ceil(…)** to round the result always up. When the bricks can be transferred with **an exact number of courses**, the division will return a whole number and there will be nothing to round. Accordingly, if not, the result of the division will be **the number of exact courses** but with a decimal fraction. The decimal part will be rounded up and we will get the required **1 course** for the remaining bricks.



In the end, we print the result on the console.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#1>.

## Problems with Simple Condition

**The second** problem of the Practical Exam covers **conditional statements and simple calculations**. Here are a few examples:

### Problem: Point on a Segment

**A horizontal segment** is placed on a horizontal line, set with the **x** coordinates of both ends: **first** and **second**. **A point** is located **on** the same horizontal line and is set with its **x coordinate**. Write a program that checks whether the point is **inside or outside of the segment** and calculates **the distance to the nearest end** of the segment.

#### Input Data

As parameters of the function we give **3 integers**:

* The first number – **the one end of the segment**.
* The second number – **the other end of the segment**.
* The point number – **the location of the point**.

All inputs are integers in the range [**-1000 … 1000**].

### Output Data

Print the result on the console:

* On the first line, print "**in**" or "**out**" – whether the point is inside or outside the segment.
* On the second line, print the distance from the point to the nearest end of the segment.

#### Sample Input and Output

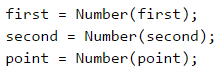
| **Input** | **Output** | **Visualization** |
| --- | --- | --- |
| 10 5 7 | in 2 |  |

| **Input** | **Output** | **Visualization** |
| --- | --- | --- |
| 8 10 5 | out 3 |  |

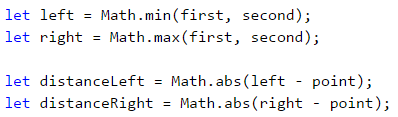
| **Input** | **Output** | **Visualization** |
| --- | --- | --- |
| 1 -2 3 | out 2 |  |

#### Hints and Guidelines

We read the input from the console.



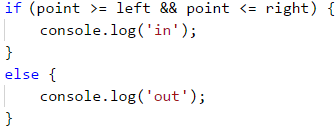
Since we do not know which **point** is on the left and which is on the right, we will create two variables to mark this. Since **the left point** is always the one with the smaller **x coordinate**, we will use **Math.min(…)** to find it. Accordingly, **the right point** is always the one with a larger **x coordinate** and we will use **Math.max(…)**. We will also find the distance from **the point x** to **the two points**. Because we do not know their position relative to each other, we will use **Math.abs(…)** to get a positive result.



The shorter of the two **distances** we will find by using **Math.min(…)**.



What remains is to find whether **the point** is on or out of the line. The point will be **on the lines** always when it **matches** one of the other two points or its x coordinate lies **between them**. Otherwise, the point is **outside the line**. After checking, we display one of the two messages, depending on which condition is satisfied.



Finally, we print **the distance** which we fine before.

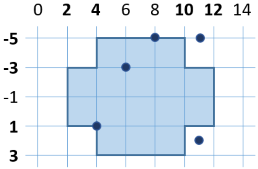


#### Testing in The Judge System

Test your solution here: [https://judge.softuni.org/Contests/Practice/Index/939#2](https://judge.softuni.org/Contests/Practice/Index/939#3).

### Problem: Point in Figure

Write a program that checks if a point (with coordinates **x** and **y**) is **inside** or **outside** of the given figure:



#### Input Data

As parameters of the function, we give **two integers**: **x** and **y**.

All inputs are integers in the range **[-1000 … 1000]**.

### Output Data

Print on the console "**in**" or "**out**" – whether the point is **inside** or **outside** the figure (the outline is inside).

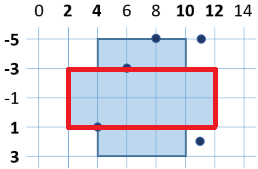
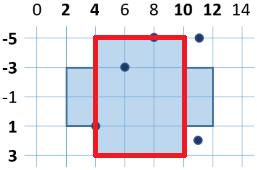
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 8 -5 | in | 6 -3 | in |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 11 -5 | out | 11 2 | out |

#### Hints and Guidelines

To find out if **the point** is inside of the figure, we will divide **the figure** into 2 rectangles:

A sufficient condition is a **point** to be located in one of them, to be in **the figure**.

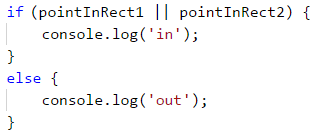
We read the input from the console:



We will initialize two variables that will mark whether **the point** is in one of the rectangles.



When printing the message, we will check whether any of the variables has accepted a value of **true**. It's enough **only one** of them to be **true** so that the point is in the figure.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#3>.

## Problems with Complex Conditions

**The third** problem of the Practical Exam includes **several nested checks combined with simple calculations**. Here are a few examples:

### Problem: Date After 5 Days

There are two numbers **d** (day) and **m** (month) that form **a date**. Write a program that prints the date that will be **after 5 days**. For example, 5 days after **28.03** is the date **2.04**. We assume that the months: April, June, September, and November have 30 days, February has 28 days, and the rest have 31 days. Months to be printed with **leading zero** when they contain a single digit (e.g. 01, 08).

#### Input Data

As parameters of the function we give **two integers**:

* An integer **d** in the range [**1 … 31**] – day. The number of the day does not exceed the number of days in that month (e.g. 28 for February).
* An integer **m** in the range [**1 … 12**] – month. Month 1 is January, month 2 is February, …, month 12 is December. The month may contain a leading zero (e.g. April may be written as 4 or 04).

### Output Data

Print a single line containing the date after 5 days in the format **day.month** on the console. The month must be a 2-digit number with a leading zero, if necessary. The day must be written without leading zero.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 28 03 | 2.04 | 27 12 | 1.01 |

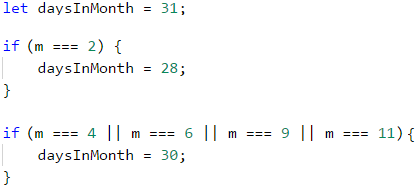
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 25 1 | 30.01 | 26 02 | 3.03 |

#### Hints and Guidelines

We take the input from the console.



To make our checks easier, we'll create a variable that will contain the **number of days** that we have in the month we set.



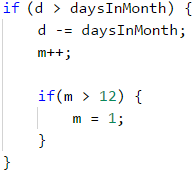
We are increasing **the day** with 5.



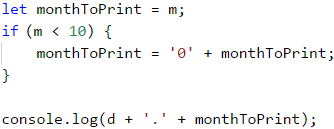
We check if **the day** has not exceeded the number of days in **the month**. If so, we must deduct the days of the month from the obtained day to calculate which day of the next month our day corresponds to.



After we have passed to **the next month**, this should be noted by increasing the initial one by 1. We need to check if it has not become greater than 12 and if it has, to adjust it. Because we cannot skip more than **one month** when we increase by 5 days, the following check is enough.



The only thing that remains is to print the result on the console. It is important to **format the output** correctly to display the leading zero in the first 9 months. This is done by creating a new variable for the month, to which to add 0 if necessary. Finally, we print the day and the new variable for the month.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#4>.

### Problem: Sums 3 Numbers

There are **3 integers** given. Write a program that checks if **the sum of two of the numbers is equal to the third one**. For example, if the numbers are **3**, **5**, and **2**, the sum of two of the numbers is equal to the third one: **2 + 3 = 5**.

#### Input Data

As parameters of the function, we give **three integers**. The numbers are in the range [**1 … 1000**].

### Output Data

* Print a text line on the console containing the solution of the problem in the format "**a + b = c**", where **a**, **b** and **c** are among the three input numbers and **a ≤ b**.
* If the problem has no solution, print “**No**” on the console.

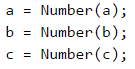
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 5 2 | 2 + 3 = 5 | 2 2 4 | 2 + 2 = 4 |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 1 5 | No | 2 6 3 | No |

#### Hints and Guidelines

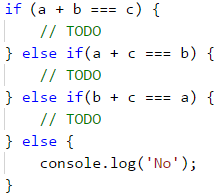
We take the input from the console.



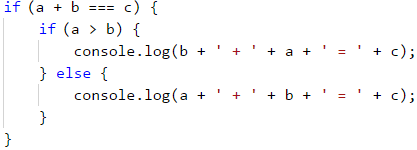
We have to check if **the sum** of a pair of numbers is equal to the third number. We have three possible cases:

* a + b = c
* a + c = b
* b + c = a

We will write **a template**, which will later be complemented by the required code. If none of the above three conditions is met, we will make our program print "**No**".



We now have to understand the order in which the **two addends** will be written in the output of the program. For this purpose, we will create **a nested condition** that checks which one of the two numbers is the larger one. In the first case, it will look like this:



Similarly, we will supplement the other two cases. The full code of the program will look like this:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#5>.

## Problems with Simple Loops

**The fourth** problem of the Practical Exam includes a **simple loop with simple logic** in it. Here are a few examples:

### Problem: Sums Step 3

There are given **n** integers **a1, a2, …, an**. Calculate the sums:

* **sum1 = a1 + a4 + a7** + … (the numbers are summed, starting from the first one with step of 3).
* **sum2 = a2 + a5 + a8** + … (the numbers are summed, starting from the second one with step of 3).
* **sum3 = a3 + a6 + a9** + … (the numbers are summed, starting from the third one with step of 3).

#### Input Data

As an input of our function, we give the array with size **n+1 (0 ≤ n ≤ 1000)**. The array will contain **the number** of the numbers **n** and **n integers** in the range [**-1000 … 1000**]: **a1, a2, …, an**.

### Output Data

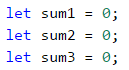
On the console, we should print 3 lines containing the 3 sums in a format such as in the example.

#### Sample Input and Output

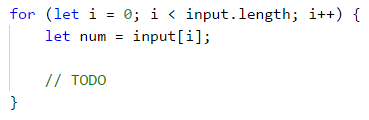
| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 2 3 5 | sum1 = 3 sum2 = 5 sum3 = 0 | 4 7 -2 6 12 | sum1 = 19 sum2 = -2 sum3 = 6 | 5 3 5 2 7 8 | sum1 = 10 sum2 = 13 sum3 = 2 |

#### Hints and Guidelines

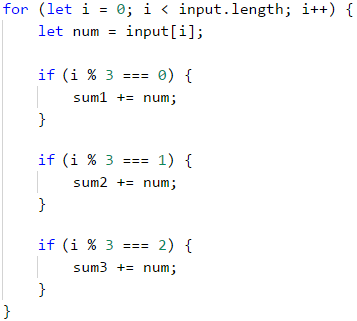
We will take **the count of numbers** (the size of the input array) and will declare **starting values** of the three sums.



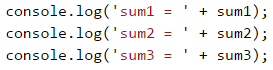
Since we do not know in advance how many numbers we will process, we will take them one at a time in **a loop** which will be repeated **n times** and we will process them in the body of the loop.



To find out which of **the three sums** we need to add the number, we will divide its **sequence number into three** and we will use **the remainder**. We'll use the variable **i** which tracks **the number of runs** of the loop, to find out which sequence number we are at. When the remainder of **i/3** is **zero**, it means we will add this number to **the first** sum, when it's **1** to **the second**, and when it's **2** to **the third**.



Finally, we will print the result on the console in the required **format**.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#6>.

### Problem: Increasing Elements

A series of **n** numbers is given: **a1**, **a2**, **…**, **an**. Calculate **the length of the longest increasing sequence** of consecutive elements in the series of numbers.

#### Input Data

We give an array with size **n+1** (**0 ≤ n ≤ 1000**) as a parameter of the function. The array will contain **the count** of the numbers **n** and **n integers** in the range [**-1000 … 1000**]: **a1**, **a2**, **…**, **an**.

### Output Data

Print on the console one number – **the length** of the longest increasing sequence.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **3** 5 2 4 | 2 | **4** 2 8 7 6 | 2 | **4** 1 2 4 4 | 3 | **4** 5 6 7 8 | 4 |

#### Hints and Guidelines

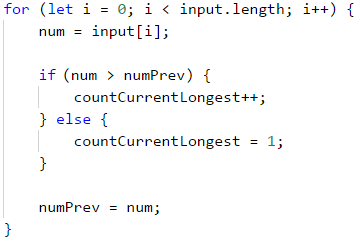
To solve this problem, we need to think in a bit **more algorithmic way**. A **sequence of numbers** is given to us, and we need to check whether each **subsequent** one will be **larger than the previous one** and if so, we count how long is the sequence in which this condition is fulfilled. Then we have to find **which sequence** of these is **the longest one**. To do this, let's create some variables that we will use during solving the problem.



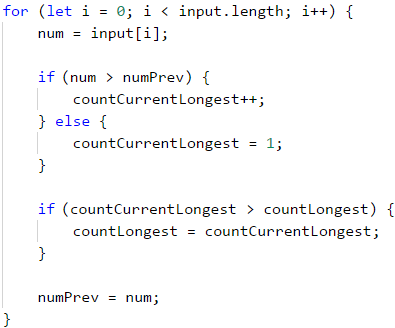
The variable **n** is **the count of numbers** we get from the console. In **countCurrentLongest** we'll keep **the number of elements** in the increasing sequence we are **currently counting**. For example, in the sequence: 5, 6, 1, 2, 3 **countCurrentLongest** will be 2 when we reach **the second element** of the counting (5, **6**, 1, 2, 3) and will become 3 when we reach **the last element** (5, 6, 1, 2, **3**) because the increasing row 1, 2, 3 has 3 elements. We will use **countLongest** to keep **the longest** increasing sequence. The other variables are **num** - the number we are **currently** in and **numPrev** - **the previous number** which we will compare with **num** to see if the row is **growing**.

We begin to run the numbers and check if the present number **a** s larger than the previous **numPrev**. If this is true, then the row **is growing**, and we need to increase its number by **1**. This is stored in the variable that tracks the length of the sequence we are currently in – **countCurrentLongest**. If the number **num** **isn't bigger** than the previous one, it means that **a new sequence** starts, and we have to start the count from **1**. Finally, after all the checks are done, **numPrev** becomes **the number** that we're **currently** using, and we start the loop from the beginning with **the next** entered **num**.

Here is a sample implementation of the algorithm described:



What remains is to see which of all sequences is **the longest**. We will do this by checking in the loop if **the sequence** we are **currently** in has become longer than **the longest one by now**. The whole loop will look like this:



Finally, we print the length of **the longest** sequence found.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#7>.

## Problems for Drawing Figures on The Console

**The fifth** problem of the Practical Exam requires **using one or several nested loops for drawing** a figure on the console. Logical reasoning, simple calculations, and conditional statements may be required. The problem tests the ability of students to think logically and invent simple algorithms for solving problems, i.e. to think algorithmically. Here are some examples of exam tasks:

### Problem: Perfect Diamond

Write a function that takes as a parameter **n** and draws **a perfect diamond** with size **n** as in the examples below.

#### Input Data

One parameter - an integer **n** in the range [**1 … 1000**].

### Output Data

The diamond should be printed on the console as in the examples below.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 | \*  \*-\*  \* | 3 | \*   \*-\*  \*-\*-\*  \*-\*   \* |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 4 | \*   \*-\*   \*-\*-\*  \*-\*-\*-\*  \*-\*-\*   \*-\*   \* | 5 | \*   \*-\*   \*-\*-\*   \*-\*-\*-\*  \*-\*-\*-\*-\*  \*-\*-\*-\*   \*-\*-\*   \*-\*   \* |

#### Hints and Guidelines

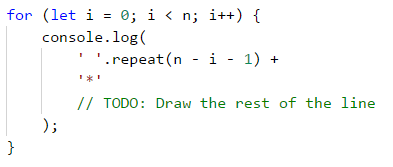
In tasks for drawing figures, the most important thing to consider is **the sequence** in which we will draw. Which items are **repeated** and with what **steps**? We can see that **the top and the bottom** parts of the diamond are **the same**. The easiest way to solve the problem is by creating **a loop** that draws **the upper part**, and then **another loop** that draws **the bottom part** (opposite to the top one).

We will read the number **n** from the parameters of the function.



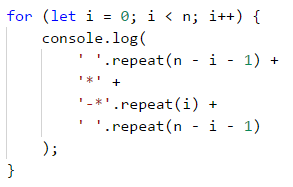
We start to draw **the upper half** of the diamond. We see that **each line** starts with some **empty spaces and \***. If we take a closer look, we will notice that **the empty spaces** are always equal to the **n - index of row - 1** (the first row is n-1, the second – n-2, etc.). We will start by drawing the number of **empty spaces** and **the first star**. Notice that we start to count from **0, no from 1**. After that, we'll only add a few times **-\*** to **finish the line**.

Here is the fragment from the code for **the upper part of the diamond**:



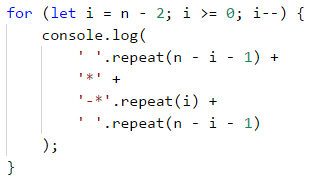
What remains is to **complete each line** with the required number of **-\*** elements. On each row we have to add **i** such **items** (on the first 1-1 -> 0, the second -> 1, etc.)

Here is the complete code for drawing **the upper part of the diamond**:



To draw **the bottom part** of the diamond, we have to reverse **the upper part**. We'll count from **n - 2** because if we start from **n - 1**, we will draw the middle row twice. Do not forget to change **the step** from **++ to --**.

Here is the code for drawing **the bottom part of the diamond**:



What remains is **to assemble the whole program** by first reading the input, printing the top part of the diamond, and then the bottom part of the diamond.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#8>.

### Problem: Rectangle with Stars

Write a function that takes as a parameter an integer **n** and draws **a rectangle** with size **n with 2 stars in the center** as in the examples below.

#### Input Data

The parameter is an integer **n** in the range [**2 … 1000**].

### Output Data

The rectangle should be printed on the console as in the examples below.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 | %%%% %\*\*% %%%% | 3 | %%%%%% % % % \*\* % % % %%%%%% |

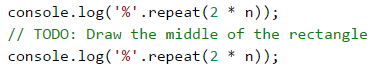
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 4 | %%%%%%%% % % % \*\* % % % %%%%%%%% | 5 | %%%%%%%%%% % % % % % \*\* % % % % % %%%%%%%%%% |

#### Hints and Guidelines

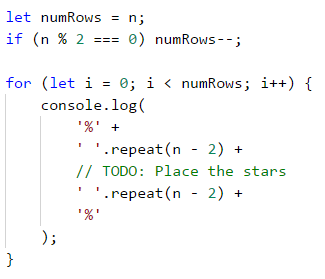
We read the input parameter of the function.



The first thing we can easily notice is that **the first and the last rows** contain **2 \* n** symbols **%**. We will start with this and then draw the middle part of the rectangle.

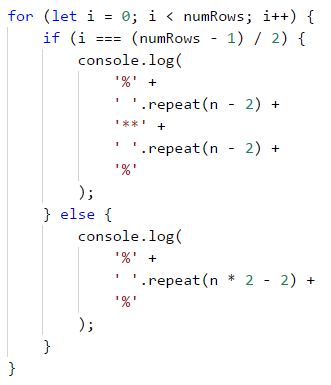


From the examples, we see that **the middle** part of the figure always has an **odd number** of rows. Note that when an **even number** is set, the number of rows is equal to **the previous odd** number (2 -> 1, 4 -> 3, etc.). We create a variable that represents the number of rows that our rectangle will have, and correct it if the number **n is even**. Then we will draw **a rectangle without the asterisks**. Each row has for **the beginning and at the end** the symbol **%** and between them **2 \* n - 2** empty spaces (the width is **2 \* n** and we subtract 2 for the two percent at the end). Do not forget to move the code for **the last line after the loop**.



We can **start and test the code so far**. Everything without the two asterisks in the middle should work correctly.

Now, **in the body** of the loop let's add the **asterisks**. We'll check if we're on the **middle row**. If we are in the middle, we will draw **the row** together **with the asterisks**, if not – we will draw **a normal row**. The line with the asterisks has **n-2 empty spaces** (**n** is half the length and we remove the asterisk and the percentage), **two stars**, and again **n-2 empty spaces**. We leave out of the check the two percent at the beginning and the end of the row.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#9>.

## Problems with Nested Loops

**The last** (sixth) problem of the Practical Exam requires using of **several nested loops and more complex logic inside them**. The problems examine participants' ability to think algorithmically and to solve non-trivial coding problems that require nested loops. Here are some examples of exam problems.

### Problem: Increasing 4 Numbers

For given pair of numbers **a** and **b** generate all four number **n1, n2, n3, n4,** for which **a ≤ n1 < n2 < n3 < n4 ≤ b**.

#### Input Data

As parameters of the function, we get two integers **a** and **b** in the range [**0 … 1000**].

### Output Data

The output contains all **numbers in batches of four**, in ascending order, one per line.

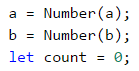
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 7 | 3 4 5 6 3 4 5 7 3 4 6 7 3 5 6 7 4 5 6 7 | 15 20 | 15 16 17 18 15 16 17 19 15 16 17 20 15 16 18 19 15 16 18 20 15 16 19 20 15 17 18 19 15 17 18 20 15 17 19 20 15 18 19 20 16 17 18 19 16 17 18 20 16 17 19 20 16 18 19 20 17 18 19 20 |

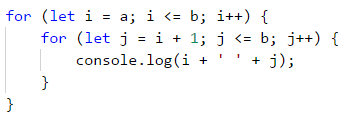
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 5 7 | No | 10 13 | 10 11 12 13 |

#### Hints and Guidelines

We read the input data from the function. We also create the additional variable **count**, which will keep track of **existing number ranges**.



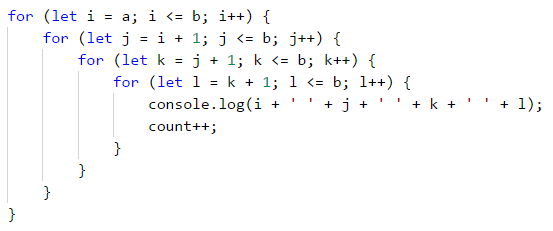
We will most easily solve the problem if we logically divide it **into parts**. If we are required to draw all the rows from a number between **a** and **b**, we will do it using **one loop** that takes all the numbers from **a** to **b**. Let's think about how to do this with a **series of two numbers**. The answer is easy – we will use **nested loops**.



We can test the incomplete program to see if it's accurate so far. It must print all pairs of numbers **i**, **j** for which **i ≤ j**.

Since each **next number** of the row must be **greater** than **the previous one**, the second loop will run around **i + 1** (the next greater number). Accordingly, if **there is no sequence** of two incremental numbers (**a** and **b** are equal), the second loop **will not be fulfilled**, and nothing will be printed on the console.

**Similarly**, what remains is to implement **the nested loops** for **four numbers**. We will add an **increase of the counter** that we initialized to know if **there is such a sequence**.



Finally, we will check if **the counter** is equal to **0** and we will print "**No**" on the console accordingly, if so.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#10>.

### Problem: Generating Rectangles

By a given number **n** and **a minimum area m**, generate all possible rectangles with integer coordinates in the range [**-n…n**] with an area of at least **m**. The generated rectangles must be printed in the following format:

**(left, top) (right, bottom) -> area**

Rectangles are defined using the top left and bottom right corners. The following inequalities are in effect:

* **-n ≤ left < right ≤ n**
* **-n ≤ top < bottom ≤ n**

#### Input Data

We get two integers as parameters of the function:

* An integer **n** in the range [**1 … 100**] – sets the minimum and maximum coordinates of a peak.
* An integer **m** in the range [**0 … 50 000**] – sets the minimum area of the generated rectangles

### Output Data

* The described rectangles should be printed on the console in a format such as in the examples below.
* If there are no rectangles for the specified **n** and **m**, then print "**No**".
* The order of rectangles in the output is not important.

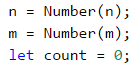
#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 2 | (-1, -1) (0, 1) -> 2 (-1, -1) (1, 0) -> 2 (-1, -1) (1, 1) -> 4 (-1, 0) (1, 1) -> 2 (0, -1) (1, 1) -> 2 | 2 17 | No |

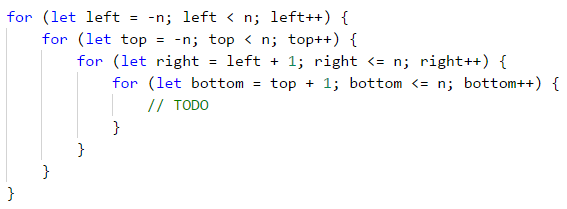
| **Input** | **Output** |
| --- | --- |
| 3 36 | (-3, -3) (3, 3) -> 36 |

#### Hints and Guidelines

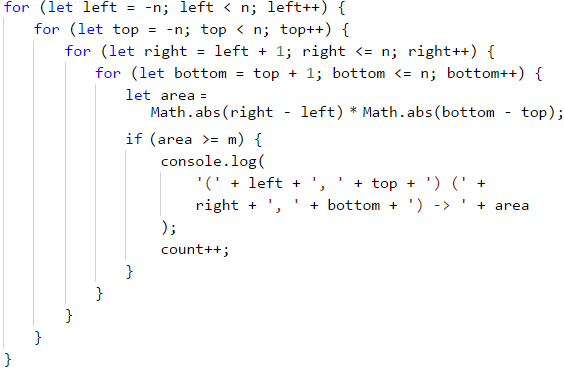
We read the input parameters of the function. We will also create a **counter**, which will store the number of rectangles found.



It is very important to be able to imagine the problem before we begin to solve it. In our case, it is required to search for rectangles in a coordinate system. The thing we know is that the **left point** will always have the coordinate **x, smaller** than **the right** one. Accordingly, **the upper one** will always have a smaller **y** coordinate than **the lower one**. To find all the rectangles, we'll have to create **a loop** similar to the previous problem, but this time, **not every next loop** will start from **the next number** because some of **the coordinates** can be equal (for example **left** and **top**).



The important thing here is knowing the corresponding coordinates so we can correctly calculate the sides of the rectangle. Now we have to find **the area of the rectangle** and check if it is **greater than** or **equal** to **m**. One **side** will be **the difference between left and right** and **the other one – between top and bottom**. Since the coordinates may be eventually interchanged, we will use **absolute values**. Again, we add **the counter** in the loop, counting **only the rectangles** we write. It is important to note that the writing order is **left**, **top**, **right**, **bottom**, as it is set in the problem's description.



Finally, we print “**No**”, if there are no such rectangles.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/939#11>.

# Chapter 8.2. Exam Preparation - Part II

In the current chapter, we will review a **practical exam**, conducted at SoftUni on December 18, 2016. The problems will give you a good overview of what you can expect at an admission exam in Programming at SoftUni. The exam covers the material studied in the current book.

## Exam Problems

Traditionally, the admission exam at SoftUni consists of **6 practical programming problems**:

* Simple problems (no conditions).
* A problem with a single condition.
* A problem with more complex conditions.
* A problem with a single loop.
* A problem with nested loops (drawing a figure on the console).
* A problem with nested loops and more complex logic.

Let's examine a **real exam topic**, the problems it contains, and their solutions.

## Problem: Distance

Write a program that calculates **what is the distance passed by a car (in kilometers)**, if we know **the initial speed** (km/h), **the initial time frame** in minutes, then the **speed is increased by 10%**, **the second time frame**, then the **speed is decreased by 5%**, and the **time until the end** of the trip. To calculate the distance, you need to **convert the minutes into hours** (for example 70 minutes = 1.1666 hours).

### Input Data

**4 arguments** are given as parameters of the function:

* **The start speed in km/h** – an integer within the range [**1 … 300**].
* **The first time in minutes** – an integer within the range [**1 … 1000**].
* **The second time in minutes** – an integer within the range [**1 … 1000**].
* **The third time in minutes** – an integer within the range [**1 … 1000**].

### Output Data

Print a number on the console: **the kilometers passed**, formatted up to the **second digit after the decimal point**.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 90 60 70 80 | 330.90 | **Distance with initial speed**: 90 km/h \* 1 hour (60 min) = **90 km** **After speed increase**: 90 + 10% = 99.00 km/h \* 1.166 hours (70 min) = **115.50 km** **After speed decrease**: 99 - 5% = 94.05 km/h \* 1.33 hours (80 min) = **125.40 km** **Total number of km passed**: **330.9 km** |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 140 112 75 190 | 917.12 | **Distance with initial speed**: 140 km/h \* 1.86 hours (112 min) = **261.33 km** **After speed increase**: 140 + 10% = 154.00 km/h \* 1.25 hours (75 min) = **192.5 km** **After speed decrease**: 154.00 - 5% = 146.29 km/h \* 3.16 hours (190 min) = **463.28 km** **Total number of km passed**: **917.1166 km** |

### Hints and Guidelines

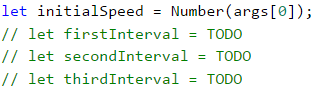
Such a description may look **misleading** and incomplete at first glance, which **adds** to the **complexity** of a relatively easy task. Let's **separate** the problem into a few **sub-problems** and try to **solve** each of them one by one, which will lead us to the final result:

* **Receiving** the input data.
* **Execution** of the main programming logic.
* **Calculation** and shaping up the final result.

**The main** part of the programming logic is to calculate what will be the **distance passed after all speed changes**. As during **execution** of the program, part of the data that we have is modified, we could **separate** the program code into a few logically separated parts:

* **Calculation** of the **distance** passed with initial speed.
* Change of **speed** and calculation of the **distance** passed.
* Last change of **speed** and **calculation**.
* **Summing up**.

On condition for **input** will be submitted **four** arguments of the function which we have to **convert to numbers** to make needed calculations. We will convert them using the **Number(...)** constructor:



In this way, we solved successfully the **first sub-problem** - receiving the input data.

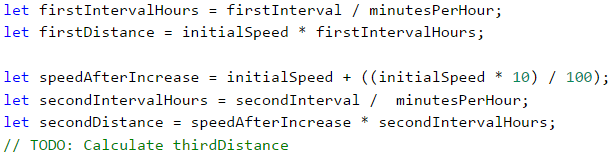
We initially **store** one **variable** that will be used multiple times. This centralization approach gives us **flexibility** and the **possibility** to **modify** the final result of the program with minimum effort. In case we need to change the value, we must do it in **only one place in the code**, which saves us time and effort:



|  |  |
| --- | --- |
|  | **Avoiding repetitive code** (centralization of the program logic) in the tasks that we examine in the present book may look unnecessary at first glance, but this approach is very important upon building large applications in a real work environment, and its exercising in an initial stage of training will help you build a quality programming style. |

We calculate the **travel time** (in hours) by **dividing the time by 60** (minutes in an hour). The **travel distance** is calculated by **multiplying the starting speed by the time passed** (in hours). After that, we change the speed by increasing it by **10%**(on condition), as per the task description. Calculating the **percentage**, as well as the following **distances** passed, is done in the following way:

* **The time frame** (in hours) is calculated by **dividing** the provided time frame in minutes by the minutes that are contained in an hour (60).
* **The distance passed** is calculated by **multiplying** the time frame (in hours) by the speed that is obtained after the increase.
* The next step is to **decrease the speed** by **5%**, as per the problem description.
* We calculate the **remaining distance** in the manner described in the first two points.



Up until now, we were able to **solve two** of the **most important sub-problems**, namely the **receiving data input** and **their processing**. What remains is to **calculate the final result**. As by the description, we are required to **format it** up to **2** symbols after the decimal point, we can do this in the following **manner**:



If you worked accurately and wrote the program using the input data given in the task description, you will be convinced that it works properly.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#0>.

## Problem: Changing Tiles

Haralambi has some **savings** that he wants to use to **change the tiles** on the bathroom floor. The **floor is rectangular**, and the **tiles are triangular**. Write a program that **calculates if his savings will be sufficient**. **The width and length of the floor are submitted**, as well as **one of the sides of the triangle with its height towards it**. We must **calculate how many tiles are needed,** to cover the floor. The **number** of tiles **must be rounded up to the higher integer** and **5 more tiles must be added** as spare tiles. Also, **we have submitted** – **the price per tile** and **the amount paid for the work** of a workman.

### Input Data

We submit 7 numbers as parameters of the function:

* **The savings**.
* **The width** of the floor.
* **The length** of the floor.
* **The side** of the triangle.
* **The height** of the triangle.
* **The price** of a tile.
* **The sum** for the workman.

**All** numbers are real numbers within the range [**0.00 … 5000.00**].

### Output Data

In this problem, our currency will be **lv**, which is BGN (Bulgarian lev). The following must be printed on the console as a **single line**:

* If the money **is sufficient**:
  + “{Remaining funds} left.”
* If the money **IS NOT sufficient**:
  + “You'll need {Insufficient funds} lv more.”

The result must be **formatted up to the second symbol** after the decimal point.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 500 3 2.5 0.5 0.7 7.80 100 | 25.60 lv left. | **Floor area** → 3 \* 2.5 = **7.5** **Tile area** → 0.5 \* 0.7 / 2 = **0.175** **Needed tiles** → 7.5 / 0.175 = 42.857… = **43 + 5 spare tiles** = **48** **Total amount** → 48 \* 7.8 + 100 (workman) = **474.4** **474.4 < 500** → **25.60 lv (BGN) left** |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 1000 5.55 8.95 0.90 0.85 13.99 321 | You'll need 1209.65 lv more. | **Floor area** → 5.55 \* 8.95 = **49.67249** **Tile area** → 0.9 \* 0.85 / 2 = **0.3825** **Needed tiles** → 49.67249 / 0.3825 = 129.86… = **130 + 5 spare tiles** = **135** **Total amount** → 135 \* 13.99 + 321 (workman) = **2209.65** **2209.65 > 1000** → **1209.65 lv (BGN) are insufficient** |

### Hints and Guidelines

The following task requires our problem to accept more input data and to perform a larger number of calculations, even though the solution is **identical**. Accepting the input data is done **familiarly**.

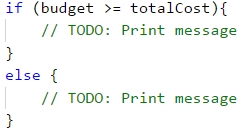
Now that we already have everything for executing the programming logic, we can move to the following part. How can we **calculate** what is the **needed** number of tiles that will be sufficient to cover the entire floor? The requirement is that tiles have a **triangular** shape, which can confuse, but practically, the task needs just **simple calculations**. We can calculate the **common part of the floor** by the formula for finding a rectangle area, as well as the **area of a single tile** using the relevant formula for the triangle area.

To calculate the **number of tiles** that are needed, **we divide the floor area by the area of a single tile** (we should not forget to add the 5 additional tiles, that were mentioned in the requirements).

|  |
| --- |
|  |



Pay attention that the requirements state that we should round up the number of tiles, obtained upon the division, up to the higher number, and then we should add 5. Find more information about the system functionality that does that: Math.ceil(…). </tr> </table> We can find the final result by **calculating the total amount** that is needed to cover the entire floor, by **summing up the tile price and the price that will be paid to the workman**, that we have from the input data. We can figure out that **the total costs** for tiles can be calculated by **multiplying the number of tiles by the price per tile**. We will find out whether the amount that we have will be sufficient by comparing the savings (based on the input data) and the total costs:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#1>.

## Problem: Flowers

A flowers shop offers 3 types of flowers: **chrysanthemums**, **roses**, and **tulips**. The prices depend on the season. In this problem, our currency will be **lv**, which is BGN (Bulgarian lev).

| **Season** | **Chrysanthemums** | **Roses** | **Tulips** |
| --- | --- | --- | --- |
| spring / summer autumn / winter | 2.00 lv./pc. 3.75 lv./pc. | 4.10 lv./pc. 4.50 lv./pc. | 2.50 lv./pc. 4.15 lv./pc. |

On holidays, the prices of all flowers are **increased by 15%.** The following **discounts** are offered:

* For purchasing more than 7 tulips in spring – **5% of the price** of the whole bouquet.
* For purchasing 10 or more roses in winter – **10% of the price** of the whole bouquet.
* For purchasing more than 20 flowers in total in any season – **20% of the price** of the whole bouquet.

**Discounts are made in the above-described order and can be combined! All discounts are valid after increasing the price on a holiday!**

The price for arranging a bouquet is always **2 lv** (BGN). Write a program that calculates the **price of a bouquet**.

### Input Data

The function receives **5 arguments**:

* **The number** of the purchased **chrysanthemums** – an integer number inside the interval of [**0 … 200**].
* **The number** of the purchased **roses** – an integer number inside the interval of [**0 … 200**].
* **The number** of the purchased **tulips** – an integer number inside the interval of [**0 … 200**].
* **The season** – [**Spring, Summer, Autumn, Winter**].
* **If the day is a holiday** – [**Y - yes / N - no**].

### Output Data

Print on the console 1 number – **the price of flowers**, formatted up to the second digit after the decimal point.

### Sample Input and Output

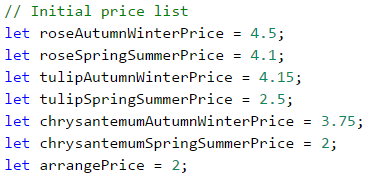
| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 2 4 8 Spring Y | 46.14 | **Price**: 2\*2.00 + 4\*4.10 + 8\*2.50 = 40.40 BGN. **Holiday**: 40.40 + 15% = 46.46 BGN. **5% discount** for more than 7 tulips in spring: 44.14 The flowers are in total 20 or less: **no discount** 44.14 + 2 **for arranging** = 46.14 BGN. |

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 3 10 9 Winter N | 69.39 | **Price**: 3\*3.75 + 10\*4.50 + 9\*4.15 = 93.60 BGN. **No holiday**: no increase in price **10% discount** for 10 or more roses in winter: 84.24 The flowers are in total over 20: **20% discount** = 67.392 67.392 + 2 **for arranging** = 69.392 BGN. |

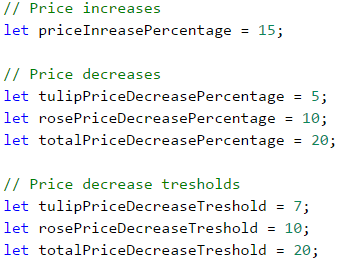
| **Input** | **Output** |
| --- | --- |
| 10 10 10 Autumn N | 101.20 |

### Hints and Guidelines

After carefully reading the requirements, we understand that once again we need to do **simple calculations**, however, this time we will need **additional** logical **conditions**. We need to pay more **attention** to the moment of **making changes** in the final price, to be able to properly build the logic of our program. Again, the bold text gives us sufficient **guidelines** on how to proceed. For a start, we will separate the already **defined** values in **variables**, as we did in the previous tasks:



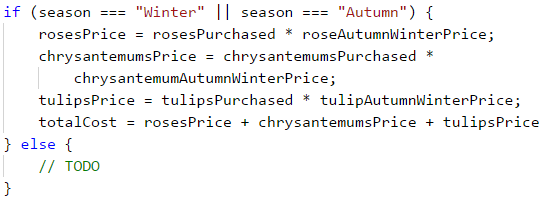
We will also do the same for the rest of the defined values:



Our next sub-task is to **read** properly **the input** data from the console. We will familiarly do that by **converting** them into numbers:



Let's think of the most appropriate way to **structure** our programming logic. By the requirements, it becomes clear that the path of the program is divided mainly into two parts: **spring / summer** and **autumn / winter**. We can do the separation by a conditional statement, by storing variables in advance for the **prices** of the individual flowers, as well as for the **final result**:



What remains is to perform **a few checks** regarding **the discounts** of the different types of flowers, depending on the season, and to modify the final result.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#2>.

## Problem: Grades

Write a program that **calculates statistics for grades** in an exam. In the beginning, the program receives the **number of students** who attended the exam and for **each student – their grade**. In the end, the program must **print the percentage of students** that have grades between 2.00 and 2.99, between 3.00 and 3.99, between 4.00 and 4.99, 5.00 or more, as well as the **average grade** of the exam.

### Input Data

The function receives **a sequence of numbers** (arguments):

* On the first line (argument) – **the number of students who attended the exam** – an integer within the range [**1 … 1000**].
* For **each student** on a separate line (argument) – **the grade on the exam** – a real number within the range [**2.00 … 6.00**].

### Output Data

Print on the console **5 lines** that hold the following information:

* "Top students: {percentage of students with a grade of 5.00 or more}%".
* "Between 4.00 and 4.99: {between 4.00 and 4.99 included}%".
* "Between 3.00 and 3.99: {between 3.00 and 3.99 included}%".
* "Fail: {less than 3.00}%".
* "Average: {average grade}".

The results must be **formatted up to the second symbol** after the decimal point.

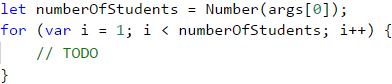
### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 10 3.00 2.99 5.68 3.01 4 4 6.00 4.50 2.44 5 | Top students: 30.00% Between 4.00 and 4.99: 30.00% Between 3.00 and 3.99: 20.00% Fail: 20.00% Average: 4.06 | 5 or more - **3 students** = 30% of 10 Between 4.00 and 4.99 - **3 students** = 30% of 10 Between 3.00 and 3.99 - **2 students** = 20% of 10 Under 3 - **2 students** = 20% of 10 The average grade is: 3 + 2.99 + 5.68 + 3.01 + 4 + 4 + 6 + 4.50 + 2.44 + 5 = 40.62 / 10 = 4.062 |

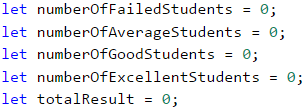
| **Input** | **Output** |
| --- | --- |
| 6 2 3 4 5 6 2.2 | Top students: 33.33% Between 4.00 and 4.99: 16.67% Between 3.00 and 3.99: 16.67% Fail: 33.33% Average: 3.70 |

### Hints and Guidelines

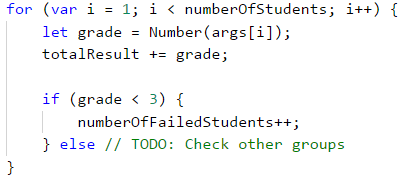
By the requirements, we see that **first**, we will receive **the number** of students, and then, **their grades**. For that reason, **first**, we will obtain **the number** of students. To process the grades themselves, we will use a **for** loop. Every iteration of the loop will read and process one grade:



Before executing the code of the **for** loop, we will create variables where we will store **the number of students** for each group: poor results (up to 2.99), results from 3 to 3.99, from 4 to 4.99, and grades above 5. We will also need one more variable, where we will store **the sum of all grades**, via which we will calculate the average grade of all students:



We run the **loop** and inside it, we **declare one more** variable, in which we will store the **currently** entered grade. The variable will be **Number** type and upon each iteration, we will check **what is its value**. According to this value, **we increase** the number of students in the relevant group by **1**, as we should not forget to also increase the **total** amount of the grades, which we also track:



We can calculate what **percentage** is taken by a particular **group of students** from the total number by **multiplying the number of students** in the relevant group by **100** and then dividing this by the **total number of students**.

**The final result** is formed in a well-known way **up to the second symbol** after the decimal point.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#3>.

## Problem: Christmas Hat

Write a program that reads from the console an **integer n** and draws a **Christmas hat** with a width of **4 \* n + 1 columns** and a height of **2 \* n + 5 rows**, as in the examples below.

### Input Data

The parameter of the function is passed **one argument - an integer n** within the range [**3 … 100**].

### Output Data

Print on the console a **Christmas hat**, exactly like in the examples.

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 4 | ......./|\....... .......\|/....... .......\*\*\*....... ......\*-\*-\*...... .....\*--\*--\*..... ....\*---\*---\*.... ...\*----\*----\*... ..\*-----\*-----\*.. .\*------\*------\*. \*-------\*-------\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*.\*.\*.\*.\*.\*.\*.\*.\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

| **Input** | **Output** |
| --- | --- |
| 7 | ............./|\............. .............\|/............. .............\*\*\*............. ............\*-\*-\*............ ...........\*--\*--\*........... ..........\*---\*---\*.......... .........\*----\*----\*......... ........\*-----\*-----\*........ .......\*------\*------\*....... ......\*-------\*-------\*...... .....\*--------\*--------\*..... ....\*---------\*---------\*.... ...\*----------\*----------\*... ..\*-----------\*-----------\*.. .\*------------\*------------\*. \*-------------\*-------------\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*.\*.\*.\*.\*.\*.\*.\*.\*.\*.\*.\*.\*.\*.\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* |

### Hints and Guidelines

In tasks requiring **drawing** on the console, most often the user inputs **an integer** that is related to the **total size of the figure** that we need to draw. As the task requirements mention how the total length and width of the figure are calculated, we can use them as **starting points**. In the examples, it is clear that regardless of the input data, we always have the **first two rows** that are almost identical.

......./|\.......  
.......\|/.......

We also notice that the **last three rows** are always present, as **two** of them are completely **the same**.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
\*.\*.\*.\*.\*.\*.\*.\*.\*  
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

By these observations, we can come up with the **formula** for the **height of the variable part** of the Christmas hat. We use the formula specified in the task to calculate the total height, by subtracting the size of the unchangeable part. We obtain **(2 \* n + 5) – 5** or **2 \* n**.

To **draw** the **dynamic** or the variable part of the figure, we will use a **loop**. The size of the loop will be from **0** to the **width** that we have by requirements, namely **4 \* n + 1**. Since we will use this formula in **a few places** in the code, it is a good practice to declare it in a **separate variable**. Before running the loop, we should **declare variables** for the **number** of individual symbols that participate in the dynamic part: **dots** and **dashes**. By analyzing examples, we can also prepare formulas for the **starting values** of these variables. Initially, the **dashes** are **0**, but we can calculate the number of **dots** by subtracting **3** from the **total width** (the number of symbols that are building the top of the Christmas hat) and then **dividing by 2**, as the number of dots on both sides of the hat is the same.

.......\*\*\*.......  
......\*-\*-\*......  
.....\*--\*--\*.....  
....\*---\*---\*....  
...\*----\*----\*...  
..\*-----\*-----\*..  
.\*------\*------\*.  
\*-------\*-------\*

What remains is to execute the body of the loop, as **after each** drawing we **decrease** the number of dots by **1** and **increase the number of dashes** by **1**. Let's not forget to draw one **star** between each of them. The sequence of drawing in the body of the loop is the following:

* Symbol string of dots
* Star
* Symbol string of dashes
* Star
* Symbol string of dashes
* Star
* Symbol string of dots

In case we have worked properly, we will obtain figures identical to those in the examples.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#4>.

## Problem: Letters Combinations

Write a program that prints on the console **all combinations of 3 letters** within a specified range, by skipping the combinations **containing a certain letter**. Finally, print the number of printed combinations.

### Input Data

The input of the program contains **exactly 3 lines** (arguments):

* A small letter from the English alphabet – between **'a'** and **'z'**.
* A small letter from the English alphabet – between the **first letter** and **'z'**.
* A small letter from the English alphabet – from **'a'** to **'z'** – as the combinations containing this letter are skipped.

### Output Data

Print on one line **all combinations** corresponding to the requirements, followed by **their number**, separated by a space.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| a c b | aaa aac aca acc caa cac cca ccc 8 | All possible combinations with letters '**a**', '**b**' and '**c**' are: aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc The combinations **containing 'b' are not valid**. **8** valid combinations remain. |

| **Input** | **Output** |
| --- | --- |
| f k h | fff ffg ffi ffj ffk fgf fgg fgi fgj fgk fif fig fii fij fik fjf fjg fji fjj fjk fkf fkg fki fkj fkk gff gfg gfi gfj gfk ggf ggg ggi ggj ggk gif gig gii gij gik gjf gjg gji gjj gjk gkf gkg gki gkj gkk iff ifg ifi ifj ifk igf igg igi igj igk iif iig iii iij iik ijf ijg iji ijj ijk ikf ikg iki ikj ikk jff jfg jfi jfj jfk jgf jgg jgi jgj jgk jif jig jii jij jik jjf jjg jji jjj jjk jkf jkg jki jkj jkk kff kfg kfi kfj kfk kgf kgg kgi kgj kgk kif kig kii kij kik kjf kjg kji kjj kjk kkf kkg kki kkj kkk 125 |

| **Input** | **Output** |
| --- | --- |
| a c z | aaa aab aac aba abb abc aca acb acc baa bab bac bba bbb bbc bca bcb bcc caa cab cac cba cbb cbc cca ccb ccc 27 |

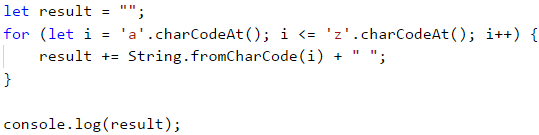
### Hints and Guidelines

By requirements, we have input data of **3 arguments**, each of which is represented by one character of the **ASCII table** (<https://www.asciitable.com/>). We could use the already defined method in JavaScript, **.charCodeAt()** through which we will receive the ASCII code of the symbol:



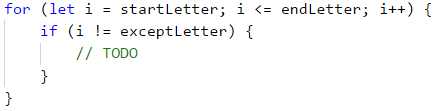
Let's think of how we can achieve the **final result**. In case the task requirement is to print all characters, from the starting to the end one (by skipping a particular letter), what should we do?

The easiest and most efficient way is to use a **loop**, by passing through **all characters** and printing those that are **different** from the **letter** that we need to skip. In JavaScript, we can go around all the symbols from 'a' to 'z' in this way:



The method **String.fromCharCode(...)** will convert the received ASCII code into a symbol. The result of running the code is all letters from **a** to **z** included, printed on a single line and separated by spaces. Does this look like the final result of our task? We must find a **way** to print **3 characters**, as required, instead of **1**. Running such a program very much looks like a slot machine. We often win in slots, if we arrange a few identical characters in a row. Let's say that the machine has space for three characters. When we **stop** on a particular **character** in the first place, the other two places will **continue** rolling characters among all possible ones. In our case, **all possible characters** are the letters from the starting to the end one, entered by the user, and the solution of our program is identical to the way a slot machine works.

We use a **loop** that runs through **all characters** from the starting to the end letter (included). On **each iteration** of the **first** loop, we run a **second** one with the same parameters (but **only if** the letter of the first loop is valid, i.e. does not match the one that we must exclude, by requirements). In each iteration of the **second** loop, we run **one** more with the **same parameters** and the same **condition**. This way we have three nested loops and in the body of **the latter** we will add the symbols to the final result:



Let's not forget that we also need to print the **total number of valid combinations** that we have found, and they must be printed on the **same line**, separated by a space. We leave this sub-task to the reader.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/940#5>.

# Chapter 9.1. Problems for Champions – Part I

In this chapter, we will offer the reader a few **more difficult tasks** that aim to develop **algorithmic skills** and acquire **programming techniques** to solve tasks with higher complexity.

## More Complex Problems on The Studied Material

We will solve together several programming problems that cover the material studied in the book, but are more difficult than the usual problems of the practical exams at SoftUni. If you want to become a **champion on the basics of programming**, we recommend this training to solve such complex tasks to make it easy for you to take exams.

## Problem: Crossing Sequences

We have two sequences:

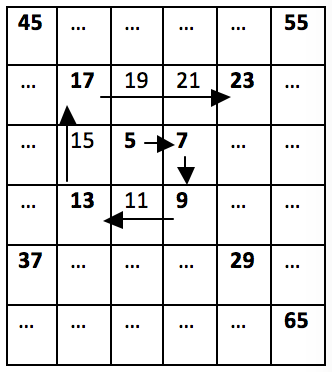
* **a sequence of Tribonacci** (by analogy with the Fibonacci sequence), where each number is **the sum of the previous three** (with given three numbers)
* a sequence generated by a **numerical spiral** defined by looping like a spiral (right, bottom, left, top, right, bottom, left, top, etc.) of a matrix of numbers starting from its center with a given starting number and incremental step, by storing the current numbers in the sequence each time we make a turn.

Write a program that finds the first number that appears **in both sequences defined in an aforementioned way**.

### Problem

Let **the Tribonacci sequence** start with **1**, **2** and **3**. This means that **the first sequence** will contain the numbers 1, 2, 3, 6, 11, 20, 37, 68, 125, 230, 423, 778, 1431, 2632, 4841, 8904, 16377, 30122, 55403, 101902, and so on.

At the same time, let the **numbers in the spiral** begin with **5**, and the spiral increases by **2** at each step.



Then **the second sequence** will contain the numbers 5, 7, 9, 13, 17, 23, 29, 37, and so on. We see that **37** is the first number to be found in the Tribonacci sequence and the spiral one, and that is the desired solution to the problem.

### Input Data

As parameters of the function, we pass **5 integers**.

* The first **three parameters** will represent **the first three numbers** in the Tribonacci sequence, positive non-zero numbers, sorted in ascending order.
* The next **two parameters** are representing **the first number** and **the step** for each cell of the matrix for the spiral of numbers. Numbers representing the spiral are positive non-zero numbers.

Input data will always be valid and will always be in the format described. There is no need to check.

### Output Data

The result should be printed on the console.

On the single line of the output, we must print **the smallest number that occurs in both sequences**. If there is no number in the range [**1 … 1 000 000**], that can be found in both sequences, then print "**No**".

### Constraints

* All numbers in the input will be in the range [**1 … 1 000 000**].
* Allowed program time: 0.3 seconds.
* Allowed memory: 16 MB.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 1 2 3 5 2 | 37 | 13 25 99 5 2 | 13 | 99 99 99 2 2 | No |

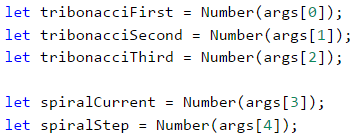
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 1 1 1 1 | 1 | 1 4 7 23 3 | 23 |

### Hints and Guidelines

The problem seems quite complicated, so we will break it into simpler sub-problems.

#### Processing The Input

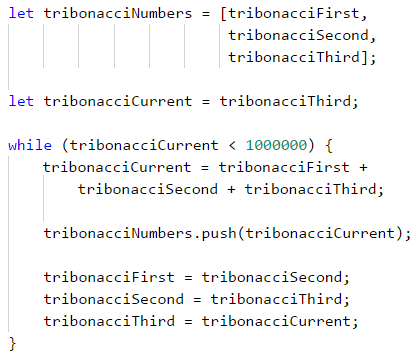
The first step in solving the problem is to read and process the input. Input data consists of **5 integers**: **3** for the Tribonacci sequence and **2** for the numerical spiral:



Once we have the input data, we need to think about how we will generate the numbers in the two sequences.

#### Generating Tribonacci Sequence

For the Tribonacci sequence, we will always **collect the previous three values** and then move the values of those numbers (the three previous ones) to one position in the sequence, i.e. the value of the first one must accept the value of the second one, and so on. When we are done with the number, we will store its value in **an array**. Since the problem description states that the numbers in the sequences do not exceed 1,000,000, we can stop generating this range at exactly 1,000,000:



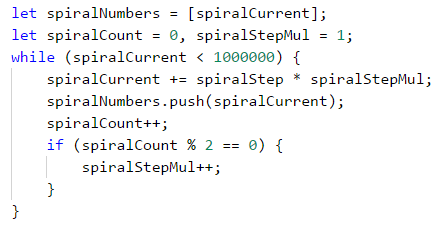
#### Generating Numerical Spiral

We need to think of **a relationship** between numbers in the numerical spiral so we can easily generate every next number without having to look at matrices and loop through them. If we carefully look at the picture from the description, we will notice that **every 2 "turns" in the spiral, the numbers we skip are increased by 1**, i.e. from 5 to 7 and from 7 to 9, not a single number is skipped, but we directly **add with the step** of the sequence. From 9 to 13 and from 13 to 17 we skip a number, i.e. we add the step twice. From 17 to 23 and from 23 to 29 we skip two numbers, i.e. we add the step three times and so on.

Thus, we see that for the first two we have **the last number + 1 \* the step**, the next two we add with the **2 \* the step** and so on. Every time we want to get to the next number of the spiral, we will have to make such calculations:



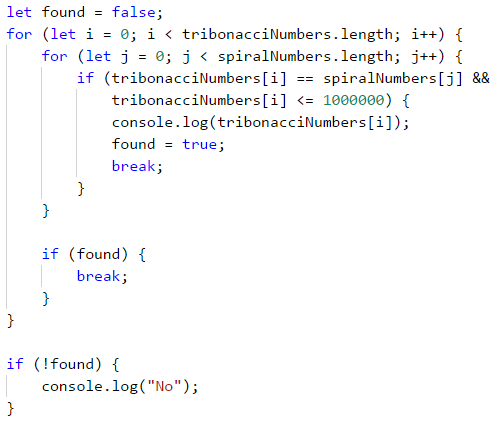
What we have to take care of is **for every two numbers, our multiplier** (let's call it "coefficient") **must increase by 1** (**spiralStepMul++**), which can be achieved with a simple check (**spiralCount % 2 == 0**). The whole code from the generation of the spiral in **an array** is given below:



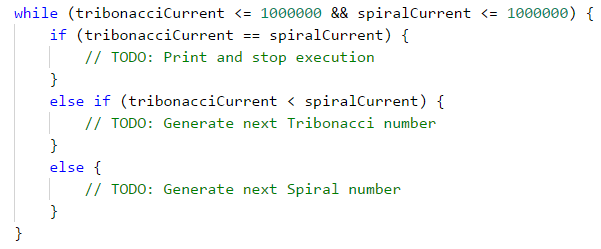
#### Finding Common Number for The Sequences

Once we have generated the numbers in both sequences, we can proceed to unite them and build the final solution. How will it look? For **each of the numbers** in the first sequence (starting from the smaller one) we will check if it exists in the other one. The first number that meets this criterion will be **the answer** to the problem.

We will do a **linear** search in the second array, and we will leave the more curious participants to optimize it using the technique called **binary search** because the second array is generated in sorted form, i.e. it meets the requirement to apply this type of search. The code for finding our solution will look like this:



The solution to the problem uses arrays to store the values. Arrays are not needed to solve the problem. There is an **alternative solution** that generates the numbers and works directly with them instead of keeping them in an array. On **every step**, we can check whether **the numbers in the two sequences match**. If this is the case, we will print the number on the console and terminate the execution of our program. Otherwise, we will see the current number of **which sequence is the smaller one and we will generate the next one where we are "lagging"**. The idea is that **we will generate numbers from the sequence that is "behind"** until we skip the current number of the other sequence and then vice versa, and if we find a match in the meantime, we will terminate the execution:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/941#0>.

## Problem: Magic Dates

**Date** is given in a "**dd-mm-yyyy**" format, e.g. 17-04-2018. We calculate **the weight of that date** by taking all of its digits, multiplying each digit with the others after it, and finally summing up all the results obtained. In our case, we have 8 digits: **17032007**, so the weight is **1\*7 + 1\*0 + 1\*3 + 1\*2 + 1\*0 + 1\*0 + 1\*7** **+** **7\*0 + 7\*3 + 7\*2 + 7\*0 + 7\*0 + 7\*7** **+** **0\*3 + 0\*2 + 0\*0 + 0\*0 + 0\*7** **+** **3\*2 + 3\*0 + 3\*0 + 3\*7** **+** **2\*0 + 2\*0 + 2\*7** **+** **0\*0 + 0\*7** **+** **0\*7** = **144**.

Our task is to write a program that finds all the **magical dates between two specific years (inclusively) corresponding to the given weight**. Dates must be printed in ascending order (by date) in the format "**dd-mm-yyyy**". We will only use the valid dates in the traditional calendar (the leap years have 29 days in February).

### Input Data

Input data consists of 3 lines:

* The first integer: **start year**.
* The second integer: **end year**.
* The third integer: **the search weight** for the dates.

Input data will always be valid and will always be in the format described. There is no need to check.

### Output Data

The result should be printed on the console as consecutive dates in **"dd-mm-yyyy" format**, sorted by date in ascending order. Each string must be in a separate line. If there are no existing magic dates, print "**No**".

### Constraints

* The start and final years are integer numbers in the range [**1900 … 2100**].
* Magic weight is an integer in the range [**1 … 1000**].
* Allowed program time: 0.25 seconds.
* Allowed memory: 16 MB.

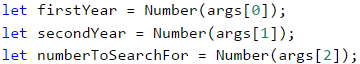
### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2007 2007 144 | 17-03-2007 13-07-2007 31-07-2007 | 2003 2004 1500 | No |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2012 2014 80 | 09-01-2013 17-01-2013 23-03-2013 11-07-2013 01-09-2013 10-09-2013 09-10-2013 17-10-2013 07-11-2013 24-11-2013 14-12-2013 23-11-2014 13-12-2014 31-12-2014 | 2011 2012 14 | 01-01-2011 10-01-2011 01-10-2011 10-10-2011 |

### Hints and Guidelines

We start with the input data. In this case, we have **3 integers**:



Having the start and the end year, it is nice to know how we will go through every date, not to worry about how many days there are in the month and whether it is a leap year, and so on.

#### Loop Through Dates

For looping through the dates, we will take advantage of the functionality that gives us the **Date** object in **JavaScript**. We will define a **start date variable** that we can do using the constructor that accepts a year, month, and day. We know the year is the starting year we read from the console and the month and the day must be January and 1st respectively. In JavaScript, the "constructor" of **Date** accepts as the first argument the year, as second argument the month (0 is January, 11 is December) and as the third argument the day of the month:

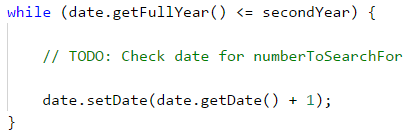


Once we have the start date, we want to create a **loop that runs until we exceed the final year** (or until we pass December 31 in the final year if we compare the full dates), increasing each day by 1 day.

To increase by one day in each rotation, we will use a method of **Date - setDate(…)**, which will add one day to the current date. The method will take care instead of us, to decide where to skip the next month, how many days there is a month, and everything around the leap years:



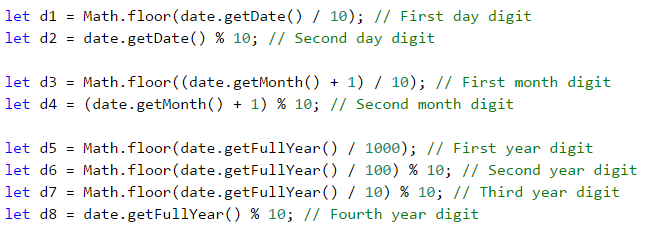
In JavaScript, we have to use the method **getFullYear()** to get the year in the same format as it is in the input data. If we use the method **getYear()** we will get the number of years passed from 1900 until the searched data, which will not help us with our problem. In conclusion, our loop should look like that:



**Caution**: we can do the same thing using **for loop**, the initialization of the date will be in the first past of the **for**, the condition stays the same, and the step is incremented with one day.

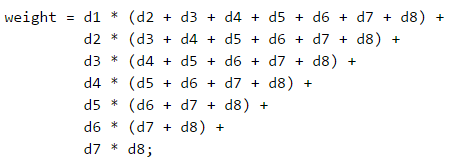
#### Calculating Date Weight

Each date consists of exactly **8 characters (digits)** – **2 for the day** (**d1**, **d2**), **2 for the month** (**d3**, **d4**), and **4 for the year** (**d5** to **d8**). This means that we will always have the same calculation every time, and we can benefit from this **to define the formula statically** (i.e. not to use loops, referring to different numbers from the date, but write the whole formula). To be able to write it, we will need **all digits from the date** in individual variables to make all the necessary multiplications. By using the division and partition operations on the individual components of the date, using the **getDay()**, **getMonth()**, and **getFullYear()** methods, we can retrieve each digit. We have to pay attention to **getMonth()**, because the method returns a number between 0 (January) and 11 (December) and we have to add **+1**, to get the months between **[1-12]**. Another thing that we have to be careful with is the division by 10 (**/ 10**), which won't be an integer division, that's why after each operation we have to round specifically to the lowest integer number by using **Math.floor(…)**:



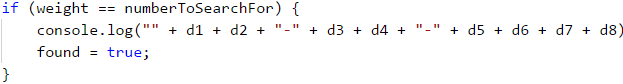
Let's also explain one of the more interesting lines here. Let's take the second digit of the year for example (**d6**). We divide the year by 100, and we take a remainder of 10. What do we do? First, we eliminate the last 2 digits of the year by dividing by 100 (Example: **2018/100 = 20**). With the remainder of 10, we take the last digit of the resulting number (**20 % 10 = 0**) and so we get 0, which is the second digit of 2018.

What remains is to do the calculation that will give us the magical weight of a given date. **To not write all multiplications** as shown in the example, we will simply apply a **grouping**. What we need to do is multiply each digit with those that follow it. Instead of typing **d1 \* d2 + d1 \* d3 + … + d1 \* d8**, we can shorten this expression to **d1 \* (d2 + d3 + … + d8)** for grouping when we have multiplication and summing up. Applying the same simplification for the other multiplications, we get the following formula:



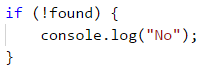
#### Printing The Output

Once we have the weight calculated for a given date, we need **to check and see if it matches the magical weight we want**, to know if it should be printed or not. Checking can be done using a standard **if** block, taking care to print the date in the correct format. Fortunately, we already have each one of the digits that we need to print **d1** until **d8**. Here we have to be careful with the data types. Since the concatenation of strings and the summation are done by the same operator, we have to convert numbers to strings or start the concatenation with an empty string:



**Caution**: as we go through the dates from the start year to the end one, they will always be arranged in ascending order as per the description.

Finally, if we have not found an eligible date, we will have a **false** value in the **found** variable and we will be able to print **No**:



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/941#1>.

## Problem: Five Special Letters

Two numbers are given: **start** and **end**. Write a program that **generates all combinations of 5 letters**, each among the sets of **{'a', 'b', 'c', 'd', 'e'}** so that the weight of these 5 letters is a number in the range **[start … end]**, inclusive. Print them in alphabetical order, in a single row, separated by a space.

**The weight of the letters** is calculated as follows:

weight('a') = 5;

weight('b') = -12;

weight('c') = 47;

weight('d') = 7;

weight('e') = -32;

**The weight of the sequence** of the letters **c1, c2, …, cn** is calculated by removing all the letters that are repeated (from right to left) and then calculating the formula:

weight(c1, c2, …, cn) = 1 \* weight(c1) + 2 \* weight(c2) + … + n \* weight(cn)

**For example**, the weight of **bcddc** is calculated as follows:

First, **we remove the repeating letters** and get **bcd**. Then we apply the formula: **1 \* weight('b') + 2 \* weight('c') + 3 \* weight('d') = 1 \* (-12) + 2 \* 47 + 3 \* 7 = 103**.

**Another example**: **weight("cadae") = weight("cade") = 1 \* 47 + 2 \* 5 + 3 \* 7 + 4 \* (-32) = -50**.

### Input Data

The input data is read from the console. It consists of two numbers:

* The number for a **start**. The number for an **end**.

Input data will always be valid and will always be in the format described. There is no need to check.

### Output Data

The result should be printed on the console as a sequence of strings, **arranged in alphabetical order**. Each string must be separated from the next one by a single space. If the weight of any of the 5 letter strings does not exist within the specified range, print "**No**".

### Constraints

* Numbers for **start** and **end** are integers in the range [**-10000 … 10000**].
* Allowed program time: 0.25 seconds.
* Allowed memory: 16 MB.

### Sample Input and Output

| **Input** | **Output** | **Comments** |
| --- | --- | --- |
| 40 42 | bcead bdcea | weight("bcead") = 41 weight("bdcea") = 40 |

| **Input** | **Output** |
| --- | --- |
| -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 |

| **Input** | **Output** |
| --- | --- |
| 200 300 | baadc babdc badac badbc badca badcb badcc badcd baddc bbadc bbdac bdaac bdabc bdaca bdacb bdacc bdacd bdadc bdbac bddac beadc bedac eabdc ebadc ebdac edbac |

| **Input** | **Output** |
| --- | --- |
| 300 400 | No |

### Hints and Guidelines

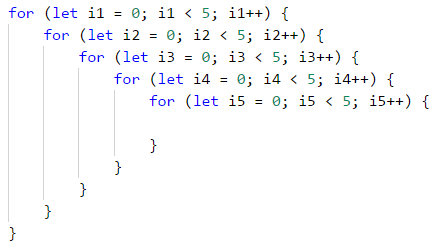
Like every problem, we start the solution by **reading and processing the input data**:



We have several main points in the problem – **generating all combinations** with a length of 5 including the 5 letters, **removing repeating letters**, and **calculating weight** for a simplified word. The answer will consist of every word whose weight is within the given range **[firstNumber, secondNumber]**.

#### Generating All Combinations

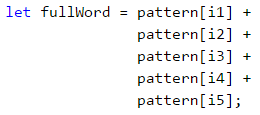
To generate **all combinations with a length of 1** using 5 symbols, we would use a **loop from 0..4**, as we want each number of the loop to match one character. To generate **any combinations of length 2** using 5 characters (i.e. "aa", "ab", "ac", …, "ba", …), we would create **two nested loops each running through the digits from 0 to 4**, as we will once again make sure that each digit matches a specific character. We will repeat this step 5 times, so we will finally have 5 nested loops with indexes **i1**, **i2**, **i3**, **i4**, and **i5**:



Now that we have all 5-digit combinations, we must find a way to "turn" the five digits into a word with the letters from 'a' to 'e'. One of the ways to do that is to **predefine a simple string that contains the letters** that we have:



**For each digit, we take the letter from the particular position.** This way, the number **00000** will become **"aaaaa"**, and the number **02423** will become **"acecd"**. We can create the 5-letter string in the following way:



**Another way:** we can convert the digits to letters by using their arrangement in the **ASCII table**. The expression **String.fromCharCode('a'.charCodeAt(0) + i)** return the result **'a'** in case **i = 0**, **'b'** in case **i = 1**, **'c'** in case **i = 2**, etc.

This way we already have generated all 5-letter combinations and can proceed with the following part of the task.

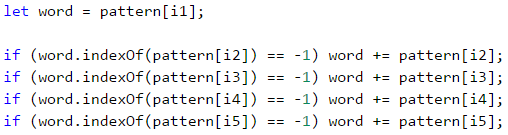
**Attention:** as we have chosen a **pattern** that takes into consideration the alphabetical arrangement of the letters, and cycles are run appropriately, the algorithm will generate the works in alphabetical order and there is no need for additional sorting before printing the output.

#### Removing Repetitive Letters

Once we have the finished string, we have to remove all the repeating symbols. We will do this by adding **the letters from left to right in a new string and each time before adding a letter, we will check if it already exists** – if it does, we will skip it and if it doesn't, we will add it. To begin with, we will add the first letter to the starting string:



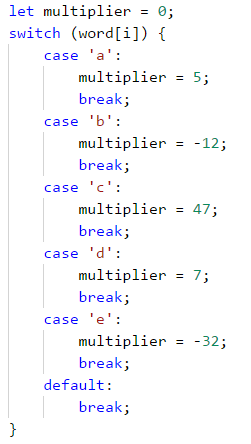
Then we will do the same with the other 4, checking each time with the following condition and the **indexOf(…)** method. This can be done with a loop by **fullWord** (leaving it to the reader for exercise), and it can be done lazily by copy-paste:



The **.indexOf(…)** method returns **the index of the particular element if it is found or** **-1** **if the item is not found**. Therefore, every time we get **-1**, it means that we still do not have this letter in the new string with unique letters and we can add it, and if we get a value other than **-1**, this will mean we already have the letter and we'll not add it.

#### Calculating Weight

Calculating the weight is simply **going through the unique word** (**word**) obtained in the last step, and for each letter, we need to take its weight and multiply it by the position. For each letter, we need to calculate what value we will multiply its position by, for example by using a **switch** construction:



Once we have the value of that letter, we should **multiply it by its position**. Because the indexes in the string differ by 1 from the actual positions, i.e. index 0 is position 1, index 1 is position 2, etc., we will add 1 to the indexes.



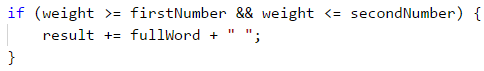
All intermediate results obtained must be added to the **total amount for each letter of the 5-letter combination**.

#### Preparing The Output

Whether a word needs to be printed is determined by its weight. We need a condition to determine if **the current weight is in the range** [**start … end**] passed to the input at the start of the program. If this is the case, we print the **full** word (**fullWord**).

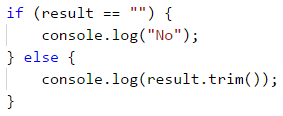
**Be careful** not to print the word with unique letters. It was only needed to calculate the weight!

The words are **separated with a space** and we'll accumulate them in an intermediate variable **result**, which is defined as an empty string at the beginning:



#### Final Touches

The condition is met **unless we do not have a single word in the entered range**. To find out if we have found a word, we can simply check whether the string **result** has its initial value (i.e., an empty string), if it does, we print **No**, otherwise we print the whole string without the last space (using the **.trim(…)** method):



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/941#2>.

# Chapter 9.2. Problems for Champions – Part II

In this chapter we will review three additional problems that belong to the category "**For Champions**", i.e. they are more complex than the rest of the problems in this book.

## More Complex Problems on The Studied Material

Before we move on to particular tasks, we must clarify that these can be solved more easily with **additional knowledge in programming with JavaScript** (functions, arrays, collections, recursion, etc.), but each solution that will be provided now only uses the material covered in this book. The goal is to learn how to construct **more complex algorithms** based on your knowledge collected up to the present moment.

## Problem: Passion Days

Lina has a real shopping passion. When she has some money, she immediately goes to the closest shopping center (mall) and tries to spend as much as she can on clothes, bags, and shoes. But her favorite thing is winter sales. Our task is to analyze her strange behavior and **calculate the purchases** that Lina does when she enters the mall, as well as the **money she has left** when the shopping is over. All prices and money are in BGN (Bulgarian levs, **lv**).

Our **first argument** of the function will be the **amount** that line has **before** she starts shopping. Our **second argument** will be a list of commands(strings), which Line will do. After that upon reading "**mall.Enter**" command, Lina enters the mall and starts shopping until the "**mall.Exit**" command is given. When Lina starts shopping **every next element** from the array will be **an action that Lina does**. Each **symbol** in the string is a **purchase or another action**. String commands contain only symbols of the **ASCII table**. The ASCII code of each sign is **related to what Lina must pay** for each of the goods. You need to interpret the symbols in the following way:

* If the symbol is a **capital letter**, Lina gets a **50% discount**, which means that you must decrease the money she has by 50% of the numeric representation of the symbol from the ASCII table.
* If the symbol is a **small letter**, Lina gets a **70% discount**, which means that you must decrease the money she has by 30% of the numeric representation of the symbol from the ASCII table.
* If the symbol is **"%"**, Lina makes a **purchase** that decreases her money in half.
* If the symbol is **"\*"**, Lina **withdraws money from her debit card** and adds 10 lv. to her available funds.
* If the symbol is **different from all of the aforementioned**, Lina just purchases without discount, and in this case, you should simply subtract the value of the symbol from the ASCII table from her available funds.

If a certain value of her purchases is **higher** than her current available funds, Lina **DOES NOT** make the purchase. Lina's funds **cannot be less than 0**.

The shopping ends when the "**mall.Exit**" command is given. When this happens, you need to **print the number of purchases made and the money** that Lina has left.

### Input Data

The input data is two arguments. Our **first argument** will indicate the **amount that Lina has before starting to purchase**. Our **second argument** will be an array of strings that will be processed in the row they are given. After you receive the command **"mall.Enter"** every next element will be a string containing **information about products/actions** that Lina wants to do. In the array, we want to perform all the commands until we receive the "**mall.Exit**" command.

Always only one "**mall.Enter**" command will be given, as well as only one "**mall.Exit**" command.

### Output Data

The output data must be **printed on the console**. When shopping is over, you must print on the console a particular output depending on what purchases have been made.

* If **no purchases have been made** – "**No purchases. Money left: {remaining funds} lv.**"
* If **at least one purchase** is made – "**{number of purchases} purchases. Money left: {remaining funds} lv.**"

**The funds** must be printed with an **accuracy of up to 2 symbols after the decimal point**.

### Constraints

* Money is a **float** number within the range: [**0 - 7.9 x 1028**].
* The number of strings between "**mall.Enter**" and "**mall.Exit**" will be within the range: [**1-20**].
* The number of symbols in each string that represents a command will be within the range: [**1-20**].
* Allowed execution time: **0.1 seconds**.
* Allowed memory: **16 MB**.

### Sample Input and Output

| **Input** | **Output** | **Comment** |
| --- | --- | --- |
| 110 mall.Enter d mall.Exit | 1 purchases. Money left: 80.00 lv. | ‘d’ has an ASCII code of 100. ‘d’ is a small letter, this is why Lina gets a 70% discount. She spends 30% of 100, which is 30 lv. After this purchase, she has: 110 - 30 = 80 lv. |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 110 ['mall.Enter' '%' 'mall.Exit'] | 1 purchases. Money left: 55.00 lv. | 100 ['mall.Enter', 'Ab', '\*\*', 'mall.Exit'] | 2 purchases. Money left: 58.10 lv. |

### Hints and Guidelines

We will separate the solution of the problem into three main parts:

* Processing of the **input**.
* **Algorithm** for solving the problem.
* Formatting the **output**.

Let's examine each of the parts in detail.

#### Processing The Input Data

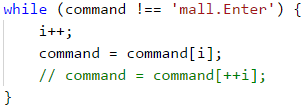
The input of our task consists of a few components:

* In the **first argument, we have all the money** that Lina has for shopping.
* In the **second argument, we have an array** that is a series of commands.

When we have the money which Lina owns we can step into processing the commands, that we get. But there is a detail that we need to take into consideration. The requirements state the following:

Our second argument will be an array of commands. They will run one after another. When we receive the command **"mall.Enter"**, our next element will be a string containing \*\*information regarding the purchases/actions that Lina wants to perform.

This is where we need to take into consideration the fact that inside **our array we should start processing commands**, but **only after we receive** the command **"mall.Enter"**. How can we do that? Using a **while** or a **do-while** loop is a good option. Here is an exemplary solution of how **to skip** all commands before processing the command **"mall.Enter"**:



We can replace this ***while*** with ***for*** loop using only the condition and step of the ***for*** loop.

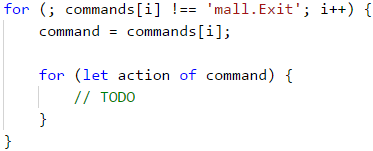
Here is the place to say that by calling **i++** after the end of the loop is used to **pass from the first command** for processing because at the end of the loop **command[i]** points exactly to **"mall.Enter"**, which mustn't be processed as action inside the mall.

#### Algorithm for Solving The Problem

The algorithm for solving the problem is a direct one – we continue **reading commands** from the console **until the command "mall.Exit" is passed**. In the meantime, we **process** each symbol (**char**) of each one of the commands according to the rules specified in the task requirements, and in parallel, we **modify the amount** that Lina has and **store the number of purchases**.

Let's examine the first two problems for our algorithm. The first problem concerns the way we read the commands until we reach the **"mall.Exit"** command. The solution that we previously saw uses a **while-loop**. The second problem for the task is to **access each symbol** of the command passed. Keeping in mind that the input data with the commands is a **string** type, the easiest way to access each symbol inside the strings is via a **foreach loop**.

This is how the code will look like:



The next part of the algorithm is to **process the symbols from the commands**, according to the following rules in the requirements:

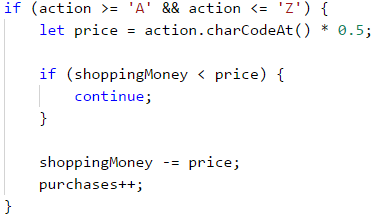
* If the symbol is a **capital letter**, Lina gets a 50% discount, which means that you must decrease the money she has by 50% of the numeric representation of the symbol from the ASCII table.
* If the symbol is a **small letter**, Lina gets a 70% discount, which means that you must decrease the money she has by 30% of the numeric representation of the symbol from the ASCII table.
* If the symbol is **"%"**, Lina purchases and that decreases her money in half.
* If the symbol is **"\*"**, Lina withdraws money from her debit card and adds 10 lv. to her available funds.
* If the symbol is **different from all of the aforementioned**, Lina just purchases without discount, and in this case, you should simply subtract the value of the symbol from the ASCII table from her available funds.

Let's examine the problems that we will be facing in the first condition. The first one is how to distinguish if a particular **symbol is a capital letter**. We can use one of the following ways:

* Keeping in mind the fact that the letters in the alphabet have a particular order, we can use the following condition **action >= 'A' && action <= 'Z'**, to check if our symbol is within the capital letters range.
* We can use the method **str.toUpperCase()** and compare if the symbol is the same as the one we will receive from **str.toUpperCase()**.

The other problem is how **to skip a particular symbol** if it is not an operation that requires more money than Lina has. This is doable using the **continue** construction.

An exemplary condition for the first part of the requirements looks like this:



**Note**: the variable “**purchases**” is of **int** type, in which we store the number of all purchases.

We believe the reader should not have difficulties implementing all the other conditions because they are very similar to the first one.

#### Formatting The Output

At the end of our task we must **print** a particular **output**, depending on the following condition:

* If no purchases have been made – "**No purchases. Money left: {remaining funds} lv.**"
* If at least one purchase is made – "**{number of purchases} purchases. Money left: {remaining funds} lv.**"

The printing operations are trivial, as the only thing we need to take into consideration is that **the amount has to be printed with an accuracy of up to 2 symbols after the decimal point**.

How can we do that? We will leave the answer to this question to the reader.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/942#0>.

## Problem: X Expression

Bonny is an exceptionally powerful witch. As her natural power is not sufficient to successfully fight vampires and werewolves, she has started to master the power of Expressions. An expression is very hard to master because the spell relies on the ability to **quickly solve mathematical expressions**.

To use an "Expression spell", the witch must know the result of a mathematical expression in advance. An **Expression spell** consists of a few simple mathematical expressions. Each mathematical expression can contain operators for **summing up**, **subtraction**, **multiplying**, and/or **division**.

The expression is solved without considering the mathematical rules for calculating numerical expressions. This means that the priority is applied according to the sequence of the operators, and not the type of calculation that they do. The expression **can contain brackets**, as **everything inside the brackets is calculated first**. Every expression can contain multiple brackets, but no nested brackets:

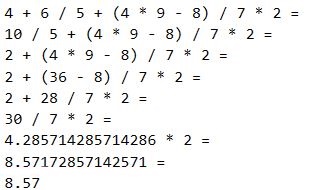
* An expression containing **(…(…)…) is an invalid one**.
* An expression containing **(…)…(…) is a valid one**.

### Problem

The expression:



is solved in the following way:



Bonny is very pretty, but not as wise, so she will need our help to master the power of Expressions.

### Input Data

The input data consists of a single text line, passed from the console. It contains a **mathematical expression for calculation**. The line **always ends with the "=" symbol**. The **"="** symbol means **the end of the mathematical expression**.

The input data is always valid and always in the described format. No need to validate it.

### Output Data

The output data must be printed on the console. The output consists of one line: the **result** of the calculated mathematical expression.

The result must be rounded up to the **second digit after the decimal point**.

### Constraints

* The expressions will consist of **a maximum of 2500 symbols**.
* The numbers of each mathematical expression will be within the range [**1 … 9**].
* The operators in the mathematical expressions will always be among **+** (summing up), **-** (subtraction), **/** (division) or **\*** (multiplying).
* The result of the mathematical expression will be within the range [**-100000.00 … 100000.00**].
* Allowed execution time: **0.1 seconds**.
* Allowed memory: **16 MB**.

### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| '4+6/5+(4\*9-8)/7\*2=' | 8.57 | '3+(6/5)+(2\*3/7)\*7/2\*(9/4+4\*1)=' | 110.63 |

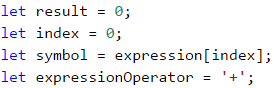
### Hints and Guidelines

As usual, we will first read and process the input, after that, we will solve the problem, and finally, we will print the result, formatted as required. In this example, our input will be 1 argument which doesn't need to be processed any further. We can directly go to solving our problem.

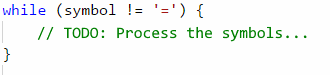
#### Algorithm for Solving The Problem

For the tasks of our problem we need to use some variables:

* One variable will store our **current result**.
* One variable will store the **index we are currently on** for our loop.
* One variable will store our **current index** which we process.
* And our final variable will store the **current operator** from our string.



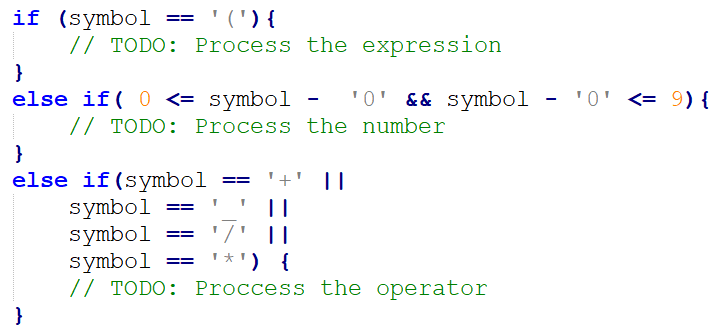
Now that we already have our starting variables, we must decide **what will be the main structure** of our program. By the requirements, we understand that **each expression ends with =**, i.e. we must read and process symbols until we reach a **=**. This is followed by an accurately written **while loop**.



Our next step is to process our **symbol** variable.

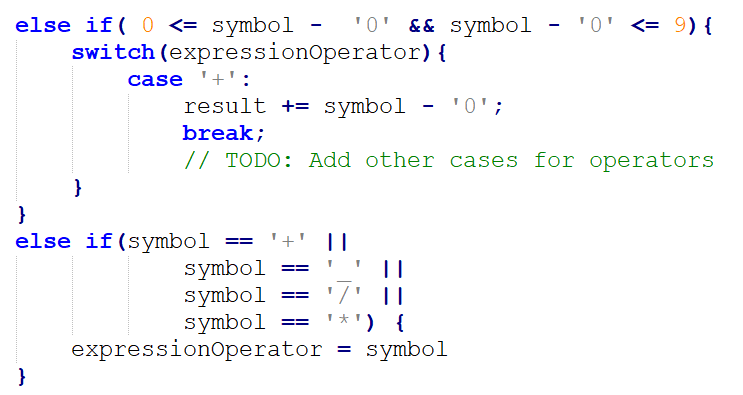
The next step is the processing of our **symbol** variable. We have 3 possible cases for it:

* If the symbol is a **start of a sub-expression placed in brackets** i.e. the found symbol is a **(**.
* If the symbol is a **digit between 0 and 9**. But how can we check this? How can we check if our symbol is a digit? We can use for assistance the **ASCII code** of the symbol, via which we can use the following formula: **[ASCII code of our symbol] – [ASCII code of the symbol 0] = [the digit that represents the symbol]**. If **the result of this condition is between 0 and 9**, then our symbol is a **number**. ( We can also use directly *'0'* and *'9'* symbols or their ***ASCII codes***.)
* If the symbol is an **operator**, i.e. it is **+**, **-**, **\*** or **/**.

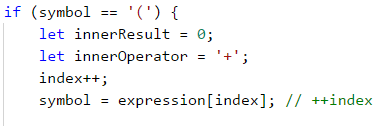


Let's examine the actions that we need to undertake in the relevant cases that we defined:

* If our symbol is an **operator**, then the only thing we need to do is to **set a new value for the expressionOperator variable**.
* If our symbol is a **digit**, then we need to **change the current result of the expression depending on the current operator**, i.e. if **expressionOperator** is a **-**, then we must **decrease the result by the numerical representation of the current symbol**. We can get the numerical representation of the current symbol via the formula that we used upon checking the condition for this case (the **[ASCII code of our symbol] – [the ASCII code of the symbol0] = [the digit that represents the symbol]**)

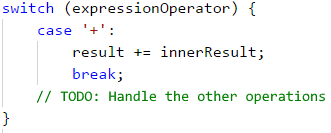


* If our symbol is a **(**, this indicates the **beginning of a sub-expression** (an expression in brackets). By definition, **the sub-expression must be calculated before modifying the result of the whole expression** (the actions in brackets are performed first). This means that we will have a local result for the sub-expression and a local operator.



After that, to **calculate the sub-expression value**, we will use the same methods that we used for calculating the main expression – we use a **while loop** to **read symbols** (until we reach an **)** symbol). Depending on whether the read symbol is a number or an operator, we modify the result of the sub-expression. The implementation of these operations is identical to the above-described implementation for calculating expressions. This is why we believe the reader will be able to easily handle it.

After finishing the result calculation for our sub-expression, we **modify the result of the whole expression** depending on the value of the **expressionOperator**.



#### Formatting The Output

The only output that the program must print on the console is the **result of solving the expression with an accuracy of up to two symbols after the decimal point**. How can we format the output this way? We will leave the answer to this question to the reader.

### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/942#1>.

## Problem: Bulls and Cows

We all know the game called "Bulls and Cows" (<https://en.wikipedia.org/wiki/Bulls_and_cows>). Upon having a particular 4-digit **secret number** and a 4-digit **suggested number**, the following rules are applied:

* If a digit in the suggested number matches a digit in the secret number and is located at the **same position**, we have a **bull**.
* If a digit in the suggested number matches a digit in the secret number but is located at a **different position**, we have a **cow**.

| **Secret number** | **1** | **4** | **8** | **1** | **Comment** |
| --- | --- | --- | --- | --- | --- |
| Suggested number | 8 | 8 | 1 | 1 | Bulls = 1 Cows = 2 |

| **Secret number** | **2** | **2** | **4** | **1** | **Comment** |
| --- | --- | --- | --- | --- | --- |
| Suggested number | 9 | 9 | 2 | 4 | Bulls = 0 Cows = 2 |

Upon having a particular secret number and the bulls and cows pertaining to it, our task is **to find all possible suggested numbers** in ascending order.

If there are **no suggested numbers** that match the criteria provided from the console, we must print "**No**".

### Input Data

Our input data consists of 3 arguments:

* The first contains **the secret number**.
* The second contains **the number of bulls**.
* The third contains **the number of cows**.

The input data will always be valid. There is no need to verify them.

### Output Data

The output data must be printed on the console. The output must consist of **a single line**, holding **all suggested numbers**, space-separated. If there are **no suggested numbers** that match the criteria provided from the console, we must **print “No”**.

### Constraints

* The secret number will always consist of **4 digits in the range** [**1..9**]. [TODO: Digits is good to be unique. If we have a secret number 2132 and suggestion 8762 We have 1 bull and 1 cow for 1 digit.]
* The number of **cows and bulls** will always be in the range [**0..9**]. [TODO: Is there a point to have cows and bulls from 0..9? When will we have 5 bulls and 7 cows? The input must be valid.]
* Allowed execution time: **0.15 seconds**.
* Allowed memory: **16 MB**.

### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 2228 2 1 | 1222 2122 2212 2232 2242 2252 2262 2272 2281 2283 2284 2285 2286 2287 2289 2292 2322 2422 2522 2622 2722 2821 2823 2824 2825 2826 2827 2829 2922 3222 4222 5222 6222 7222 8221 8223 8224 8225 8226 8227 8229 9222 |

| **Input** | **Output** |
| --- | --- |
| 1234 3 0 | 1134 1214 1224 1231 1232 1233 1235 1236 1237 1238 1239 1244 1254 1264 1274 1284 1294 1334 1434 1534 1634 1734 1834 1934 2234 3234 4234 5234 6234 7234 8234 9234 |

| **Input** | **Output** |
| --- | --- |
| 1234 3 1 | No |

### Hints and Guidelines

Because our input comes directly as an argument of our function we can do the followings steps to complete our task:

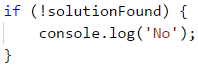
* We will generate all possible **four-digit combinations** (candidates for verification).
* For each generated combination we will calculate **how many bulls** and **how many cows** it has according to the secret number. Upon matching the needed bulls and cows, we will **print the combination**.

#### Algorithm for Solving The Problem

Before starting to write the algorithm for solving our problem, we must **declare a flag** that indicates whether a solution is found:

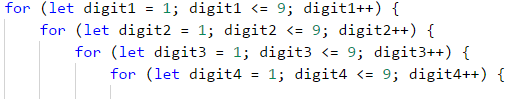


If after finishing our algorithm this flag is still **false**, then we will print **No** on the console, as specified in the requirements.

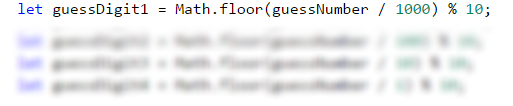


Let's start analyzing our problem. What we need to do is **analyze all numbers from 1111 to 9999**, excluding those that contain zeroes (for example **9011**, **3401**, etc. are invalid). What is the easiest way to **generate** all these **numbers**? We will **use nested loops**. As we have a **4-digit number**, we will have **4 nested loops**, as each of them will generate **an individual digit in our number** for testing.

An alternative solution is to go through all the numbers from 1111 to 999 with a loop and ignore every number with '0' inside it. But this will change the code, which we will do below.

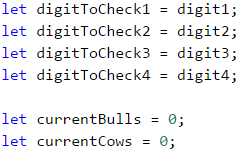


Thanks to these loops, **we have access to every digit** of all numbers that we need to check. Our next step is to **separate the secret number into digits**. This can be achieved very easily using **a combination of integer division and modular division**.



How? When we divide we get a decimal number. We can use Math.floor(…) then divide it with a remainder operator(%) by 10, or we can cast the number with parseInt(…). In the example above we remove the fractional part.

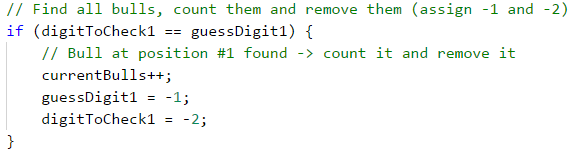
Only two last steps remain until we start analyzing how many cows and bulls there are in a particular number. Accordingly, the first one is the **declaration of counter variables** in the nested loops, to **count the cows and bulls** for the current number. The second step is to make **copies of the digits of the current number** that we will analyze, to prevent problems upon working with nested loops, in case we make changes to them.



We are ready to start analyzing the generated numbers. What logic can we use? The easiest way to check how many cows and bulls there are inside a number is via a **sequence of if-else conditions**. Yes, this is not the most optimal way, but to stick to what is covered in the current book, we will use this approach.

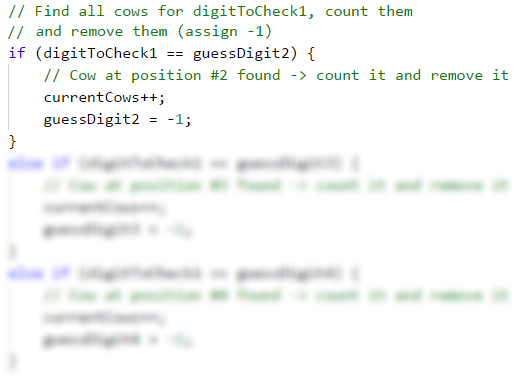
What conditions do we need?

The condition for the bulls is very simple – we check whether the **first digit** of the generated number matches the **same digit** in the secret number. We remove the digits that are already checked to avoid repetitions of bulls and cows.



We repeat the action for the second, third, and fourth digits.

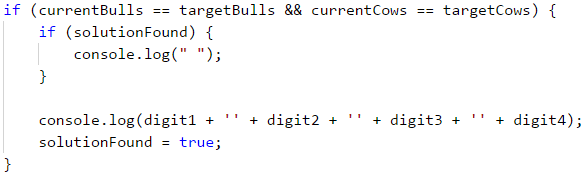
Our conditional statement for cows can be done by this method - we check if the **first digit** from the generated number **matches with the second**, **the third**, or **the fourth** digit of the secret number. We can combine all conditional statements inside one condition, know that in the three cases we have a cow, but we won't know which digit to remove. That's why we type it one by one:



After that, we sequentially check whether the **second digit** of the generated number **matches the first one**, the **third one**, or the **fourth digit** of the secret number; whether the **third digit** of the generated number matches the **first one**, the **second one**, or the **fourth digit** of the secret number; and finally, we check whether the **fourth digit** of the generated number matches the **first one**, the **second one** or the **third digit** of the secret number.

#### Printing The Output

After completing all conditions, we just need to **check whether the bulls and cows in the currently generated number match the desired bulls and cows read from the console**. If this is true, we print the current number on the console.



### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/942#2>.

# Chapter 10. Functions

**JavaScript** is a well-known **functional language** for programming. As the name hints, **functions** are an extremely important part of the language.

In this chapter, we will introduce you to **functions** and you will learn their **definition**, as well as what their **base concepts** are while working with them. You will learn why it's a **good practice** to use them, how to **define** and **declare** them. We will also introduce you to the **parameters** and **return value of a function**, as well as how to use that return value. In the end, we will peek at **best practices** when using functions.

## What is a "Function"?

Until this moment, we have found out that while **writing** a programming code for an exercise, **separating** the exercise into different **parts**, favors us a lot. Every part is responsible for a **corresponding action** and by doing so it's **easier** to solve the problem, and the **readability** and the debugging of the code are better.

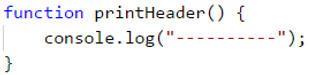
A block of code designed to perform a particular task and which we have separated logically is called **function**. **Functions – slices of code, that are named** by us in a specific way. They can be called numerous times when we need them, and they will be run that many times as we want to.

One **function** can be **called** that many times, as we think is needed for solving a problem. That **saves** us from repeating the same code and **reduces** the opportunities of making a mistake while editing the repeated code.

### Simple Functions

Simple functions are responsible for running a specific **action**, that will **help** us solve a problem. Those actions can be printing a string on the console, doing a conditional statement, doing a loop, etc.

Let's see an **example of a simple function**:



This **function** has the task to print a header, which is a series of the symbol **-**. Because of this, the name of the function is **printHeader**. The round brackets **(** and **)** are always after the name, no matter how we have named the functions. Later we will take a look at how we have to name a function we are working with. For now, we will only say that the **name of the function must define the action** that is doing.

The **body** of the function consists of **programming code**, which is located between the curly brackets **{** and **}**. Between them, we place code, that will solve our problem, described by the name of the function.

### Why Should We Use Functions?

Up to this moment, we have found out that, functions help us with **separating long exercises into smaller parts**, which leads to a **simple solution** of the corresponding problem. This makes our program, not just well structured, **easy readable** but also more understandable.

Using functions we **escape repeating** of programming code. **Repeating** code is a **bad practice** because it makes **harder maintenance** for the programmer to do which leads to errors. If one part of code exists in our program more than once and we have to fix it, we will have to change every occurrence of repeating code. The probability of us forgetting one of the repeated places is high, which will lead to incorrect behavior of our program. This is exactly why it's a **good practice** to define a fragment that will be used **more than once** as a **separate function**.

Functions offer us a **good method** to use **code several times**. With solving more and more exercises, we will conclude that using already defined functions saves us a lot of time and effort.

## Declaring Functions

In JavaScript language we can **define** functions everywhere, using the same way we define variables. Declaring represents the **registration of a function** inside a program and to be recognized inside it.

JavaScript isn't a **strongly typed** language. That's why when we **declare a function** it doesn't have a type(string, number, array, etc.), which other programming language methods have.

There are two ways, to declare a function in JavaScript - **function declaration** and **Function expression**.

### Function Declaration

With the next example, we will take a look at the required elements inside a function, using **function declaration**.



* **Keyword function**. We start by using the keyword **function**, with which we define that there will be declared a function. We name it **keyword** because it is reserved in the JavaScript language. We can't have a variable that is named **function** exactly because it is reserved.
* **Name of the function**. The name of the function is **defined by us** and we must never forget that it must **define the task** it is doing. In the example, the name **getSquare** tells us that the task of this function is to find the area of a square.
* **List of parameters**. We declare them between **(** and **)** brackets, which we type after the name of the function. Here we list a series of **parameters**, which the function will use. We can have **only one** parameter, **more than one** parameter or we can leave it empty. If there are no parameters we will only type the brackets **()**. In the current example, the parameter is only the **n**.
* **Body of the function**. It is declared between **{** and **}** brackets which we type after the closing bracket **)**. In our **body of the function**, we define all operations, that we want our function to do, **using code**. We also describe the **algorithm** by which the function will solve the given problem. We achieve the **logic** of the function. In the current example, we calculate the area of the square using **n \* n**.

When declaring functions it is important to follow the **sequence** of the fundamental elements - first **Keyword function** then **Name of function**, **List of parameters** surrounded by round brackets **()**, **Body of the function** surrounded by curly brackets **{}**.

### Function Expression

With the next example, we will take a look at the required elements in the declaration of an **expression function**. It is similar to **function declaration** which we have already viewed and it can be said that it's a **set** of **declaring a variable** and **declaring a declaration function**.



* **Keyword let**. We will start with using the **keyword let**, with which we will declare a declaration of a variable.
* **Name of the variable**. The name of the variable is **decided by us**. In the example, the name is **getSquare** which tells us that the task of this function is to calculate the area of the square.
* **Declaration of a function**. Using the same structure that we have learned in **function declaration** - first **Keyword function** then **Name of function**, **List of parameters** surrounded by round brackets **()**, **Body of the function** surrounded by curly brackets **{}**. The difference, in this case, is that **Name of a function** is not necessary, but it is recommended to get used to writing the name. In the example, the program will work without problems even if we don't type **getSquareFunc**. If we don't type the name, the function will become **anonymous**.

When we declare a variable in the body of a function (using the keyword **let** or **const**). We call it a **local** variable for the function. The scope in which it exists and can be used is from the row it is defined to the end of the closing bracket **}** of the function. That scope is called **variable scope**.

### Function Declaration or Function Expression

The difference between **Function declaration** and **Function expression** is very simple. All functions declared by **function declaration** are loaded in the memory of a program before it is started. However, when we are using **function expression** the program will know and run the function only when it has come to it.

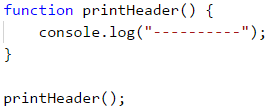
In theory, this means that we can **call a function** that is declared with **function declaration** even before it was declared in the previous rows. If we try to use **function expression** the program will **throw an error** that the function is not declared yet.

## Invoking a Function

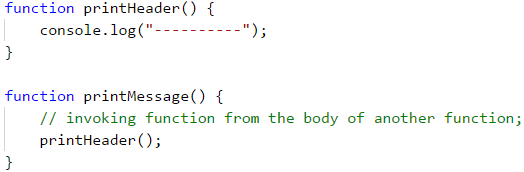
Invoking a function is the **start of the execution of a code** that is located inside the body of a function. We call it by typing the **name** of the function followed by **()** and **;** to end the row. Here is an example:



This function can be called from **different points** in our program. One of the ways to be called is to be called from the **global scope**.



A Function can be invoked from the **body of another function** which is **not** the global scope of our program.



A Function can be called from **its own body**. This is called **recursion** and you can find more about it on [Wikipedia](https://en.wikipedia.org/wiki/Recursion_(computer_science)) or google it.

### Problem: Blank Receipt

Write a function, that prints an empty cash receipt. The function must call another three functions: one to print the title, one for the main part, and the last for the bottom part.

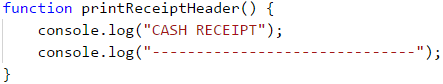
| **Part of cash receipt** | **Text** |
| --- | --- |
| Upper part | CASH RECEIPT ------------------------------ |
| Middle part | Charged to\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Received by\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |
| Bottom part | ------------------------------ (c) SoftUni |

#### Sample Input and Output

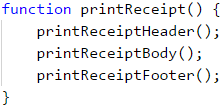
| **Input** | **Output** |
| --- | --- |
| None | CASH RECEIPT ------------------------------ Charged to\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Received by\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ ------------------------------ (c) SoftUni |

#### Hints and Guidelines

Our first step is to create a function for **creating a title**. We should give it a short descriptive name. For example: **printReceiptHeader**. In the body we will write the following code:

 In a like manner, we will create another two functions **to print the middle part** of the receipt (body) **printReceiptBody** and **to print the bottom part** of the receipt (footer **printReceiptFooter**.

After this we will set **one more function\*** which will call the other three functions which we have written so far:



In the end, we will **invoke** **printReceipt** from the global scope of our program:



#### Testing in The Judge System

The program with a total of four functions that call one another is ready and we can **run and debug it**. After that, we will send it for test in the Judge system: <https://judge.softuni.org/Contests/Practice/Index/943#0>.

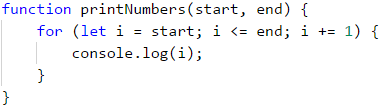
## Functions with Parameters

When we are dealing with a given task the function we are using needs **additional information** which the result depends on. Exactly this information are the **parameters of the functions** and the behavior of a program depends on.

### Using Parameters Inside a Function

If the function requires **input data**, it is passed inside **()** brackets. The order of **function parameters** must match the order of the **function arguments** when declaring and invoking the function. Parameters can be **zero, one, or more**. When declaring the parameters we divide them with a comma **,**.

**Declaring** function **printNumbers(...)** and the **list** of **parameters** which the programs need to work correctly will look like this:



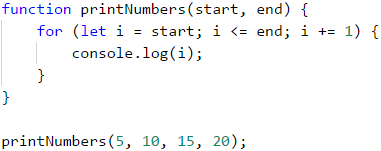
After this, we **invoke** the function, and we give it the **corresponding function arguments**:



When **declaring function parameters** we should check if every parameter has a **name**. It's also important when we invoke a function we should always pass **values** in the order that they were declared. In the previous example, the variable **start** will be passed to the first element (in our case the number 5). The variable **end** will have the next number which is 10.

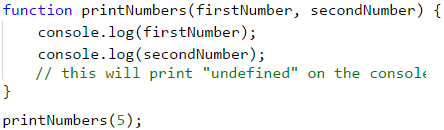
It's important to point out that in the programming language **JavaScript** declaring a function with a given **number of parameters** doesn't force us to invoke a function with the **same number of parameters**. We can invoke the function by giving it **more** or **fewer** parameters than needed and it won't cause an error.

Let's look at this example:



In this example, we call the function **printNumbers(...)** and we give 4 instead of the **declared** 2 parameters. All unnecessary parameters will be ignored. These are the numbers 15 and 20. They won't go to the function because the function doesn't have a declared parameter that will get them.

Let's look at another example:



In this example, we call the function **printNumbers(...)** but this time we give only 1 parameter instead of the **declared** 2 parameters. All parameters that are not **set as values** will automatically get **undefined** values. In our case the variable.

### Problem: Sign of Integer Number

Create a function that checks if a number is a positive or negative number.

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 2 | The number 2 is positive. |
| -5 | The number -5 is negative. |
| 0 | The number 0 is zero. |

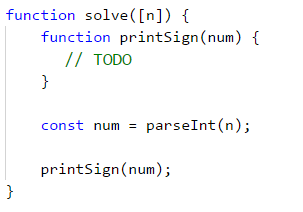
#### Hints and Guidelines

Our first step is **declaring** a function and giving it describing name - **printSign**. This function will only have one parameter:



Our next step is **implementing** logic by which our program will check if the number is positive or negative. From the example, we can see that there are 3 cases: if the number is bigger and smaller than zero or if it's zero. We will make three conditional statements in the body of our function.

Our next step is to call the function we have created:



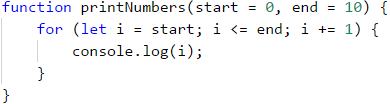
#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#1>.

### Optional Parameters

**JavaScript** programming language supports **optional** parameters. They allow **skipping** of parameters when calling a function. We declare them by **providing default values** in the description of the parameter.

Our next example shows the use of optional parameters:



**printNumbers(...)** can be invoked in different ways:



When we are not **setting value** for the parameter, he will **get the value** that we have given him when declaring the function.

### Problem: Printing Triangle

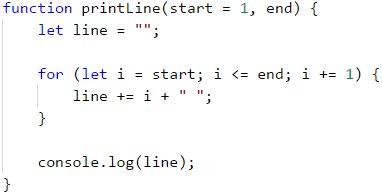
Create a function that will create a triangle as shown in the example.

#### Sample Input and Output

| **Input** | **OutPut** | **Input** | **OutPut** |
| --- | --- | --- | --- |
| 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 |

#### Hints and Guidelines

We choose a name according to the task it will do. For example, **printLine** and we implement it:



From drawing on console exercises we remember that it's a good practice to **divide the figure into different parts**. We will divide the triangle into 3 parts - upper, middle, and bottom.

Our next step is to print the **upper body** of the triangle with a loop:



After that, we will print the **middle part**:



In the end, we will print the **bottom part** from the triangle but this time with a reverse loop.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#2>.

### Problem: Draw a Filled Square

Draw a square with side **n**, as shown in the example.

#### Sample Input and Output

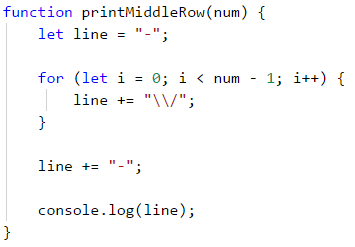
| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 4 | -------- -\/\/\/- -\/\/\/- -------- | 5 | ---------- -\/\/\/\/- -\/\/\/\/- -\/\/\/\/- ---------- |

#### Hints and Guidelines

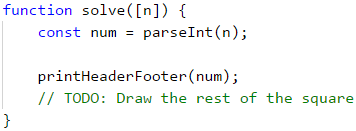
We create a function that prints the first and last row because they are the same. We shouldn't forget that we have to give him a **corresponding name** and set as a **parameter** the length of the side. We will use the built-in function **repeat(...)**:



Our next step is to create a function that will draw on the console the middle rows. Again we set a descriptive name such as **printMiddleRow**.



In the end, we invoke the declared functions to draw the whole square:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#3>.

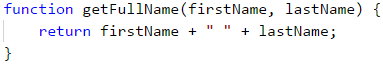
## Return Result From a Function

Up to this point, we have viewed functions that do a specific task. For example printing a text, number, or a figure on the console. There is another type of function that can **return** a **result** from their task. We can return a result of the multiplication of two numbers. We will look at this type of function in this part.

### The Operator return

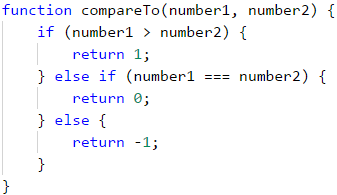
To get a result from a function we use the operator **return**. He must be **used inside the body** of a function. The **return** will tell the program to **stop the execution** of the function and **return** a corresponding value. This value is defined after the **return** word.

In the example underneath we have a **function** that gets the first and last name as **parameters**. Then it combines them and returns the full name.



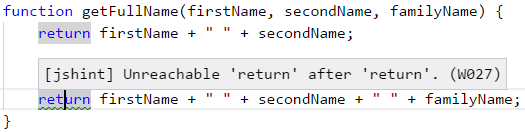
**There are cases** in which **return** can be called from different places inside the function, but only if the **conditions** are met.

In the example below, we have a function that compares 2 numbers and **returns** the result which can be **-1**, **0**, or **1** if the first argument is smaller, equal, or bigger than the second argument. The function uses the operator **return** on 3 different places. It returns different values depending on the logic of the code.



#### Code After a return is Unreachable

When the **return** operator is located inside a conditional statement such as **if**, after the statement in the same block, we must **not** have rows with code because Visual Studio Code will display a warning telling us that it had found an **unreachable** code.



The operator **return** can be also used without a **specific value**. In this case, the function will be **terminated** and the return value will be **undefined**.

|  |  |
| --- | --- |
|  | In programming, we can't have 2 times the operator return one after another because the first return won't allow us to use the second. Sometimes programmers joke with the phrase ***type* return; return; *and let's go home***”, to explain that the logic of the program is wrongly typed. |

### Using The return Value of a Function

After a function is executed and has returned a value, we can use the value in several ways.

The first is to **assign the result to a variable**:



The second is to be used inside **an expression**:



The third is to **pass** the result from one function to **another function**:



### Problem: Calculate Triangle Area

Write a function that finds the area of a triangle by given side and altitude and then returns the area.

#### Sample Input and Output

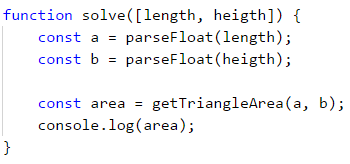
| **Input** | **Output** |
| --- | --- |
| 3 4 | 6 |

#### Hints and Guidelines

We **Create** a function with a descriptive name.



Our next step is to **call the new** function and **record the returned value inside another variable**.



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#4>.

### Problem: Math Power

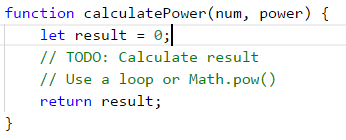
Write a function that calculates and returns the result the power of a number.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 2 8 | 256 | 3 4 | 81 |

#### Hints and Guidelines

Our first step is again to create a function that will get 2 parameters (base and exponent(power)). Then the function will return the result.



After we have done our calculations, all we are left to do is invoke the function.

#### Testing in The Judge System

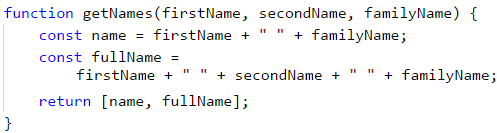
Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#5>.

### Functions, Returning Multiple Values

In practice, there are some cases in which we need a function that returns more than one element as a result. In the **JavaScript** language, there are 2 ways of doing that. They are with **destruction** and by **returning an object**.

#### Destruction

When we want a function to return **more than one value**, we use the keyword **return** and after that, we list all the values we want to return between **[**, **]** brackets:



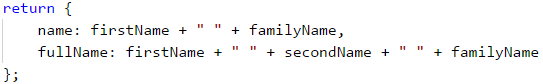
After that to get the returned values using the square brackets we list a new set of variables that will get the returned values. We must follow the order we have returned them in the function.



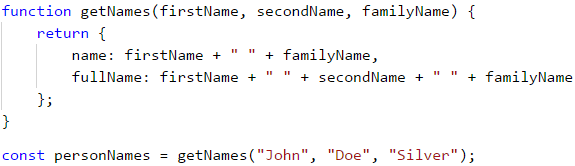
In the upper example, the variable **name** will get the value "**John Doe**", which is the first of the returned function of **getNames**, and *fullName* ***will get "***John Silver Doe\*" which is second.

#### Objects

This way is very similar to the previous one. The only difference is that we don't just **list** all the values which we want to return. We give them **names**. **Objects** are extremely important and a big part of the **JavaScript** language. For now, it's enough for us to know that they are declared with curly brackets **{** **}** and we type the **name** of the value (it's called a key). After the **name**, we type **:** followed by a value. We divide different **key-value** pairs with **,**.



In this example, we return an object which holds 2 values - **name** and **fullName**.



Here the variable **personNames** will get all the returns values. By calling **.name** and **.fullName** we get the returned values:



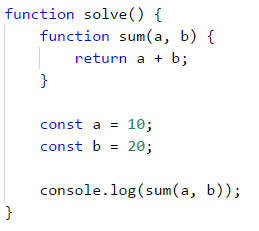
## Variants of a Function

In many programming languages, one function can be declared as **different variants** with the same name but different parameters. This is known as **method overloading**. For good or bad **JavaScript**, language doesn't support this.

When we declare **two or more functions with the same names**, our program will use the **last declared**. Declaring the second function with the same name removes the old function and overrides the new one in that place.

## Nested Functions

Let's take a look at this example:



### What is a Local Function?

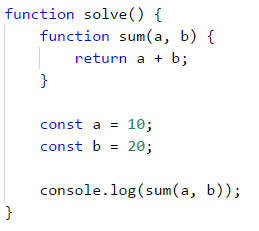
We see that in the previous example, **solve()** function has **another** declared function **sum()**. This **nested** function is called a **local** function. Local functions can be declared in every other function.

### Why Should We Use Local Functions?

With time and practice we will find out that when we type code, we often need functions, which we might need only once or another function is becoming too long. We have already said that when one function has too many rows of code it is very hard to read, support, and understand. In those cases, we can declare another function that we will use even only once. This helps the code to be cleaner and reduces the chance of making a mistake in the programming code.

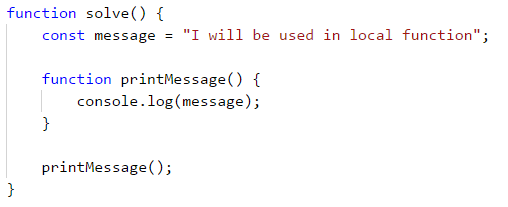
### Declaring Local Functions

Let's look again at the previous example



In this example **sum()** is local function because it's nested inside **solve()** function. This means that **sum()** can be called **only** inside **solve()** function because it's **only declared inside it**.

Local functions have access to variables, which are declared in the same or upper level from them. Our next example shows how this happens.



This feature of nested functions makes them convenient helpers when solving a task. They save time and code by not having to pass values to parameters and variables.

## Naming Functions. Best Practices For Working With Functions.

In this part, we will take a look at some of the **best practices** for writing functions that are connected to an arrangement of a code and its structure.

### Naming Functions

When we name a function it's recommended to use **logical names**. This is good because every function must **correspond** to a specific part of our problem. We must take into consideration the **task** which the function will do. This is why it's a good practice **for the name to describe the purpose of the function**.

It is required for the name of the function to start with a **small letter** and to be a verb or a combination of a verb and a noun. Formatting the names must be done following **Lower Camel Case** convention - **every word except the first to start with an upper letter**. Round brackets **(** and **)** are always after the name of the function.

Every function must do an independent task and the name must describe its role.

Here are some examples of **correctly** named functions:

* **findStudent**
* **loadReport**
* **sine**

Some examples of **badly** named functions:

* **method1**
* **doSomething**
* **handleStuff**
* **sampleMethod**
* **dirtyHack**
* **FindStudent**
* **LoadReport**

If we can't come up with a good name it's probably because our function does more than one job or it doesn't have a clear task. We must think of a way to divide the function into different functions.

### Naming Parameters of Functions

When naming **parameters** of functions, the same rules of naming functions apply to them. The only difference is that the name of the parameters should be nouns or a set of a noun and an adjective. It's a **good practice** that the name of the parameter should **indicate** what type of unit is used when working with it.

Here are some examples of **correctly** named parameters of functions:

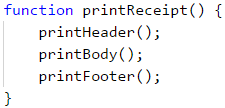
* **firstName**
* **report**
* **speedKmH**
* **usersList**
* **fontSizeInPixels**
* **font**

Some examples of **incorrectly** named parameters:

* **p**
* **p1**
* **p2**
* **populate**
* **LastName**
* **last\_name**

### Good Practices for Working With Functions

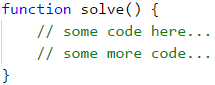
We must keep in mind that a function must do **only one** specific **task**. If this can't be achieved, then we must come up with a way to **divide** the functionality into different parts. As we have already said the name must be clear and descriptive. Another **good practice** is to **avoid** functions that are longer than our screen. If this happens it's recommended to **split** the function into smaller ones as shown in the example below.



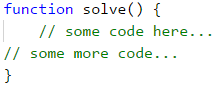
### Structure and Formatting of The Code

When writing functions we must keep in mind to follow a correct **indication** (move more inward blocks of the code).

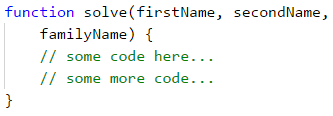
Here are some examples of **correctly** formatted JavaScript code:



Some examples of **incorrectly** formatted JavaScript code:



When the headline row of the function is **too long**, it's recommended to be split into several rows, and every row after the first should be moved 2 tabulations to right (for clear readability).



Another good practice is to **leave an empty row** between functions, after loops, and conditional statements. You should always try to **avoid** writing **long rows and complicated expressions**. Over time you will find out that improves the readability and saves time.

We recommend you to always **use curly brackets for the body of conditional statements and loops**. The brackets don't just improve the readability but also reduce the possibility of making a mistake.

## What Have We Learned from This Chapter?

In this chapter we have learned base concepts about working with functions:

* Learned that **the goal** of a function is to **split** programs with many rows to code with smaller and shorter parts.
* Comprehended the idea of the **structure** of functions and how to **declare** and **invoke** them by their name.
* Looked into different examples of functions with **parameters** and how to use them in our program.
* Learned what is **signature** and **return value** of the function as well as what is the role of the **return** operator.
* Studied **good practices** for working with functions, how to name them and their parameters and how to format our code and others.

## Problems

To improve what we have learned we will solve a few exercises. In them, it is required to write functions with specific functionality and then call it with values as shown in the example.

### Problem: Hello, Name!

Write a function that takes a name as a parameter and prints on the console "Hello, {name}!".

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| Peter | Hello, Peter! |

#### Hints and Guidelines

Define a function **printName(name)** and implement it. Write a function **solve(...)** which receives as input name of a person and calls **printName** function.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#7>.

### Problem: String Repeater

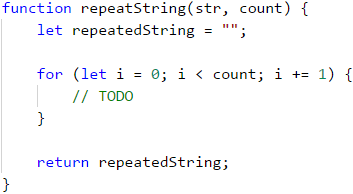
Create a function **repeatString(str, count)**, which takes parameters of type **string** and an integer **n** and returns the string, repeated **n** times. After this print the result on the console.

## Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| str 2 | strstr | roki 6 | rokirokirokirokirokiroki |

## Hints and Guidelines

In the function below, inside the loop, append the input string to the result, that you will finally return:



Keep in mind that **in JavaScript concatenating strings in loops leads to bad performance** and is not recommended.

## Testing in the Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#9>.

### Problem: Min Method

Define a function **GetMin(int a, int b)** and implement it, after which invoke it from the function **solve(...)** as shown below. To find the minimum of three numbers, first, find the minimum of the first two and then the minimum of the result and the third number:

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 1 2 3 | 1 | -100 -101 -102 | -102 |

#### Hints and Guidelines

Define function **getMin(int a, int b)** and implement it, after that call it from **solve(...)** as shown in the example below. To find the minimum of the tree numbers find first the minimum of the first and second value. Then find the minimum of the result of the two and the third number:

function solve([num1, num2, num3]) {

let min = getMin(getMin(num1, num2), num3);

}

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#8>.

### Problem: String Repeater

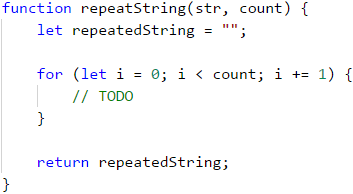
Write a function **repeatString(str, count)** which gets as parameters string variable **str** and number **n** then returns the string repeated **n** times. After that print the result on the console.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| str 2 | strstr | roki 6 | rokirokirokirokirokiroki |

#### Hints and Guidelines

Write function and add the input string to the result in the for loop:



#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#9>.

### Problem: Nth Digit

Write function **findNthDigit(number, index)** which gets number and index then print the Nth digit of the number (starting to count from left to right from 1). After that print the result to the console.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 83746 2 | 4 | 93847837 6 | 8 | 2435 4 | 2 |

#### Hints and Guidelines

To perform the algorithm we will use a **while** loop. While the number isn't 0 we will check if the index matches our input value. If it matches we will return the number of the index (**number % 10**). If it doesn't match anything we will remove the last digit of the number (**number = number / 10**). We must follow which digit we check in the conditional statement (starting from left to right from 1). When we find the digit we will return the index.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#10>.

### Problem: Integer to Base

Write a function **integerToBase(number, toBase)**, which takes as parameters an integer and a base of a numeral system and returns the integer converted to the given numeral system. After this, the result should be printed on the console. The input number will always be in the decimal numeral system, and the base parameter will be between 2 and 10.

## Sample Input and Output

| **Input** | **Output** | **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- | --- | --- |
| 3 2 | 11 | 4 4 | 10 | 9 7 | 12 |

## Hints and Guidelines

To solve the problem, we will declare a string, in which we will keep the result. After this, we need to do the following calculations to convert the number.

* Calculate **the remainder** of the number, divided by the base.
* **Insert the remainder** at the beginning of the string.
* **Divide** the number to the base.
* **Repeat** the algorithm until the input integer equals 0.

Write the missing logic in the function below:

function integerToBase(number, toBase) {

string result = "";

while (number !== 0) {

// implement the missing conversion logic

}

return result;

}

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#11>.

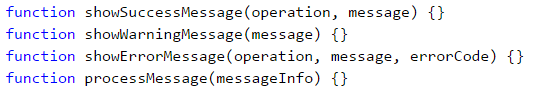
# Problem: Notifications

Write a function **solve(...)**, which takes an integer **n** and **n input messages** and prints **n output messages**, based on the input. For each message read a few lines. Each message starts with **messageType**: “**success**”, “**warning**” or “**error**”:

* When **messageType** is “**success**” read **operation** + **message** (each from a new line).
* When **messageType** is a “**warning**” read-only **message** (from a new line).
* When **messageType** is “**error**” read **operation** + **message** + **errorCode** (each from a new line).

Print on the console **each read message** formatted depending on its **messageType**. After the headline of the message print as much **=**, **as the length** of the said **headline** and print **an empty line** after each message (to understand in detail look at the examples).

The problem should be solved by defining four functions: **showSuccessMessage()**, **showWarningMessage()**, **showErrorMessage()** and **processMessage()**, so that only the last function is invoked by the **Main()** function:



#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 4 error credit card purchase Invalid customer address 500 warning Email not confirmed success user registration User registered successfully warning Customer has not email assigned | Error: Failed to execute credit card purchase. ============================================== Reason: Invalid customer address. Error code: 500.  Warning: Email not confirmed. =============================  Successfully executed user registration. ======================================== User registered successfully.  Warning: Customer has no email assigned. ========================================= |

#### Hints and Guidelines

Define and implement the four shown functions. After that call **processMessage(...)** from the main **solve(...)** function.



In **processMessage()** read the type of message from the console and according to the read, type read the rest of the data (one, two, or three more lines). After that invoke the function for printing the given type of message.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#12>.

### Problem: \* Numbers to Words

Write a function **letterize(number)**, which reads an integer and prints it in words in English according to the conditions below:

* Print in words the hundreds, the tens, and the ones (and the eventual minus) according to the rules of the English language.
* If the number is larger than **999**, you must print "**too large**".
* If the number is smaller than **-999**, you must print "**too small**".
* If the number is **negative**, you must print "**minus**" before it.
* If the number is not built up of three digits, you shouldn't print it.

#### Sample Input and Output

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 3 999 -420 1020 | nine-hundred and ninety nine minus four-hundred and twenty too large | 2 15 350 | fifteen three-hundred and fifty |

| **Input** | **Output** | **Input** | **Output** |
| --- | --- | --- | --- |
| 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 |

#### Hints and Guidelines

We can first print **the hundreds** as a text – (the number / 100) % 10, after that **the tens** – (the number / 10) % 10 and at the end **the ones** – (the number % 10).

The first special case is when the number is exactly **rounded to 100** (e.g. 100, 200, 300, etc). In this case we print "one-hundred", "two-hundred", "three-hundred" etc.

The second special case is when the number formed by the last two digits of the input number is **less than 10** (e.g. 101, 305, 609, etc...). In this case, we print "one-hundred and one", "three-hundred and five", "six-hundred and nine" etc.

The third special case is when the number formed by the last two digits of the input number is **larger than 10 and smaller than 20** (e.g. 111, 814, 919, etc). In this case, we print "one-hundred and eleven", "eight-hundred and fourteen", "nine-hundred and nineteen" etc.

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#13>.

### Problem: \* String Encryption

Write a function **encrypt(char letter)**, which encrypts a given letter in the following way:

* It takes the first and the last digit from the ASCII code of the letter and concatenates them into a string, which will represent the result.
* At the beginning of the string, which represents the result, we will insert the symbol which matches the following condition:
  + ASCII code of the letter + the last digit of the ASCII code of the letter.
* After that at the end of the string, which represents the result, you concatenate the character which matches the following condition:
  + ASCII code of the letter - the first digit of the ASCII code of the letter.
* The function should return the encrypted string.

Example:

* j → **p16i**
  + ASCII code of **j** is **106** → First digit – **1**, last digit – **6**.
  + We concatenate the first and the last digit → **16**.
  + At **the beginning** of the string, which represents the result, concatenate the symbol, which you get from the sum of the ASCII code + the last digit → 106 + 6 → 112 → **p**.
  + At **the end** of the string, which represents the result, concatenate the symbol, which you get from subtracting the ASCII code – the first digit → 106 - 1 → 105 → **i**.

Using the function shown above, write a function **solve(...)** which takes **a sequence of characters**, **encrypts them**, and prints the result on one line. The input data will always be valid. The Main function must read the data given by the user – an integer **n**, followed by a character for each of the following **n** lines. Encrypt the symbols and add them to the encrypted string. In the end, as a result, you must print **an encrypted string** as in the following example.

**Example**:

* S, o, f, t, U, n, i → V83Kp11nh12ez16sZ85Mn10mn15h

#### Sample Input and Output

| **Input** | **Output** |
| --- | --- |
| 7 S o f t U n i | V83Kp11nh12ez16sZ85Mn10mn15h |

| **Input** | **Output** |
| --- | --- |
| 7 B i r a H a x | H66<n15hv14qh97XJ72Ah97xx10w |

#### Hints and Guidelines

Our variable **result** in which we will save the value of the result we will give the initial value **""**. We must recur a loop **n** times so that in each iteration we will add the encrypted symbol to the result string.

To find the first and the last digit of the ASCII code, we will use the same algorithm that we used to solve "Integer to Base".

#### Testing in The Judge System

Test your solution here: <https://judge.softuni.org/Contests/Practice/Index/943#14>.

# Chapter 11. Tricks and Hacks

In the current chapter, we are going to see some tricks, hacks, and techniques, which will make our work with **JavaScript** easier in the **Visual Studio Code** IDE. In particular, we will see:

* How to properly **format our code**.
* Conventions for **naming elements in the code**.
* Some **keyboard shortcuts**.
* Some **code snippets**.
* Techniques to **debug our code**.

## Code Formatting

The right formatting of our code will make it **easier to read and understand** in case someone else needs to work with it. This is important because in practice we will need to work in a team with other people and it is highly important to write our code in a way that our colleagues can **quickly understand** it.

There are some defined rules for correct formatting of the code, which are collected in one place and are called **conventions**. The conventions are a group of rules, generally accepted by the programmers using a given language, which are massively used. These conventions help to build norms in given languages - what is the best way to write and what are the **good practices**. It is accepted that if a programmer follows them then his code is easy to read and understand.

The **JavaScript** language is created by **Brendan Eich** as part of the development of one of the first browsers **Netscape**. The basic constructions and basic syntax are intentionally **similar to Java**, to reduce the learning effort. Moreover, similar conventions are even used to write and format the code. You should know that, even if you do not follow the imposed conventions, the code will **work** (as long as it is written correctly), but simply **will not be easy to understand**. Of course, this is not fatal on a base level, but the faster you get used to writing quality code, the better.

The rules that are used for writing **JavaScipt** can be found in many sources. The official rules, etc. **JavaScript code conventions** are very well described in the article "**Coding style**" in the documentation of "Mozilla": <https://developer.mozilla.org/en-US/docs/Mozilla/Developer_guide/Coding_Style>. It is important, to note that in the examples, we have given so far and will give in the future in this book, we are guided mainly by it.

For code formatting **curly braces {}** to be opened on the same line and to be closed just below the construction, to which they refer, as in the example below.

if (someCondition) {

console.log("Inside the if statement");

}

You can see that the command **console.log(…)** in the example is **offset by 4 white spaces(one tab)**, which is also recommended by the documentation. If given construction with curly brackets is offset by one tab, then **the curly brackets {}** must be at **the beginning of the construction**, as in the example below:

if (someCondition) {

if (anotherCondition) {

console.log("Inside the if statement");

}

}

This is an example of a **badly formatted code** according to the accepted conventions for writing code in **JavaScript**:

if(someCondition)

{

console.log("Inside the if statement");}

The first thing we can notice is the **curly braces {}**. The first (opening) bracket must be **exactly next to the if condition**, and the second (closing) bracket - **under the command console.log(…), on a separate blank line**. In addition, the command inside the **if** construction must be **offset by 4 white spaces(one tab)**. Immediately after the keyword **if** and before the condition of the check, a **space** is left.

The same rule applies to **for loops, as well as any other constructions with curly braces {}**. Here are some more examples:

**Correct:**

for (let i = 0; i < 5; i++) {

console.log(i);

}

**Wrong:**

for(let i=0;i<5;i++)

{

console.log(i);

}

For your comfort, there are **keyboard shortcuts in Visual Studio Code**, which we will explain later in this chapter, but for now, we are interested in the following combinations for formatting **the code in the whole document**:

* For Windows [**Shift + Alt + F**]
* For Mac [**Shift + Option + F**]
* For Ubuntu [**Ctrl + Shift + I**]

Let's use **the wrongly formatted example** from earlier:

for(let i=0;i<5;i++)

{

console.log(i);

}

If we press [**Shift + Alt + F**], which is our combination to format **the entire document**, we will get code formatted according to **generally accepted JavaScript conventions**. However, automatic formatting does not affect the naming of our variables, which we have to take care of ourselves. It is accepted for variables to **begin** always **with a lowercase letter** and **contain lowercase letters**, with **each subsequent word** in them **beginning with a capital letter** (this naming is also known as the **camelCase** convention).

* We must be aware of uppercase and lowercase letters because **JavaScript makes a difference** between them. For example **age** and **Age** are different variables.
* Variable names **cannot** match a JavaScript keyword, for example, **let** is an invalid variable name. The official or, etc. keyword phrases are just words that are **part of the syntax of JavaScript** and for this reason, they are reserved and cannot be used as names of our variables. Through these words, we have the opportunity to build our programs. As an example of such words can be given the already used: **for**, **while**, **do**, **if**, **else**, **let**, etc. You can see a full list of these reserved phrases here: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Lexical_grammar#Keywords>

|  |  |
| --- | --- |
|  | Although using the symbol \_ in the manes of variables is allowed, in JavaScript it is not recommended and is considered a bad style of naming. |

Here are some examples for **well-named** variables:

* **firstName**
* **age**
* **startIndex**
* **lastNegativeNumberIndex**

Here are some examples for **badly named variables**, even though the names are correct according to the JavaScript language:

* **\_firstName** (start with '\_')
* **last\_name** (contains '\_')
* **AGE** (written in uppercase)
* **Start\_Index** (starts with an uppercase letter and contains '\_')
* **lastNegativeNumber\_Index** (contains '\_')

At a first look all these rules can seem meaningless and unnecessary, but with time passed and experience gaining you will see the need for conventions for writing quality code, to be able to work more easily and faster in a team. You will understand that the work with a code, which is written without complying with any rules for code quality, is annoying.

## Shortcuts in Visual Studio Code

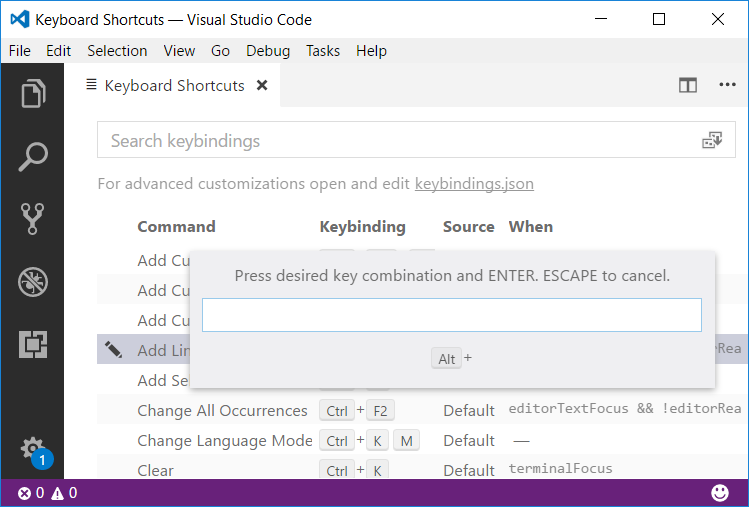
In the previous section, we mentioned some of the combinations, that are used for formatting code. One of them [**Shift + Alt + F**] is used for **formatting the whole code in a file**, and others did the same thing but on a different operating system. These combinations are called **shortcuts** and now we will give more thorough information about them.

Shortcuts are **combinations**, that give us the ability to perform some actions **easier and faster**, and each software development environment has its shortcuts, although most are repetitive. We will now look at some of the **shortcuts** in **Visual Studio Code**. The listed keyboard shortcuts work for sure and have been tested on Windows. The idea is to show you that this exists, it is easy to use and when needed, you can always find what you need for every operating system.

| **Combination** | **Action** |
| --- | --- |
| [**CTRL + F**] | The combination **opens the search window**, by which we can **search in our code**. |
| [**CTRL + /**] | **Comments** part of our code and accordingly **removes the comment** which is already commented. |
| [**CTRL + Z**] | **Brings back one change** (so-called Undo). |
| [**CTRL + Y**] | The combination is opposite of [**CTRL + Z**] (the so-called Redo). |
| [**Shift + Alt + F**] | **Formats the code** according to the default conventions. |
| [**CTRL + Backspace**] | **Deletes** the word to the left of the cursor. |
| [**CTRL + Del**] | **Deletes** the word to the right of the cursor. |
| [**CTRL + K S**] | **Saves** all files in the project. |
| [**CTRL + S**] | **Saves** the current file. |

More about the **shortcuts in Visual Studio Code** can be found here: <https://code.visualstudio.com/shortcuts/keyboard-shortcuts-windows.pdf>.

If you feel confident enough about your shortcut skills, open **Visual Studio Code** and press [**CTRL + K + S**] (note that this is different from [**CTRL + K S**], where **Ctrl** and **K** are pressed simultaneously, and **S** after that), which will open a window in the development environment itself, which contains a complete list of all possible keys combinations in the world of **Visual Studio Code**. Moreover, even from there, you will be able to make **changes** to existing keyboard shortcuts:



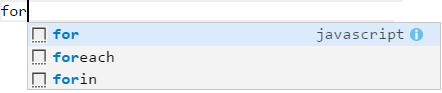
Do not hesitate, apply what you have learned now and use the keyboard shortcuts that you think will help you in writing your programs!

## Code Snippets (Code Snippets)

In **Visual Studio Code** there are the so-called **code snippets** (code snippets), which write a block of code by using a code template. This useful option is not enabled by default. You have to activate it yourself from [**File -> Preferences -> Settings**] (or just **[Ctrl + Comma]**), which opens window called **User Settings**. These are your settings, that you can easily change. Just add the following line between the opening and closing curly braces on the right side of the screen:

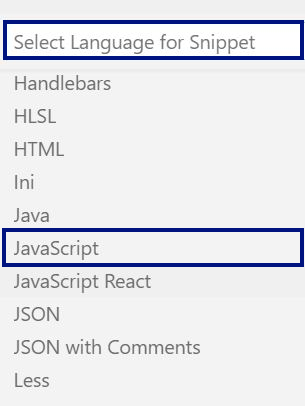
"editor.tabCompletion": true

After doing this, when you type "**for**" and press **[Tab]** + **[Tab]**, the code of **complete for loop** is automatically generated in the body of our program. This is called "unfolding a code snippet". The snippet "**if**" + **[Tab]** + **[Tab]** works similarly. On the figure below you can see the "**for**" snippet in action:

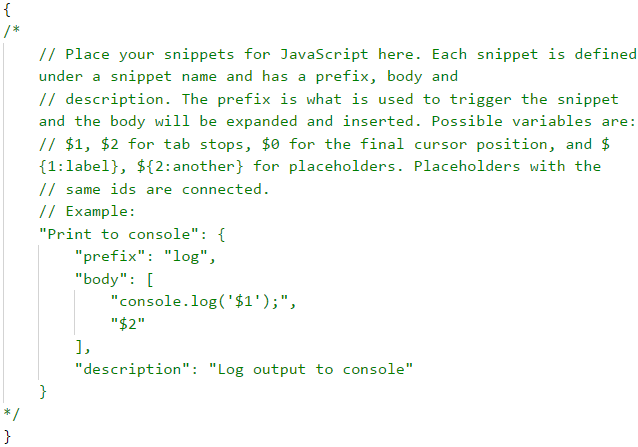


### Creating Your Code Snippet

In this section, we are going to show you how to **make your code snippet**. We will see **how to make a code snippet** for **json** object. For a start, we will go to [**File -> Preferences -> User Snippets**], then a window will open from which you can choose for which programming language you will create a snippet, as shown in the picture:



**We choose JavaScript** from the drop-down menu and a window with a name will open **JavaScript.json**. This extension **json** is a special format for saving data, which is imposed in the ways of data transfer and storage. In addition, the **json** format can be used in our programs, as we will discuss later. The file looks like this:



The example we see by default is generating code to write to the console using the **log** key prefix. This code is just an example, and this template is embedded, but if it wasn't, it would look like the example.

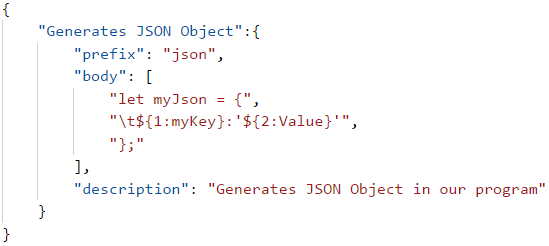
In this example, you see a lot of unknown things, but do not worry, we will get familiar with them later. We now focus on the part **"Print to console":** and the code between **opening and closing curly braces {}**. What we see inside the parentheses is the content of a template. . Each template must contain a **prefix**, which is the short text that after pressing [**Tab**] + [**Tab**] will generate the template code in your program.

The second thing your template should have is a **body**, this is the most complex part of the template. This is the **code that will be generated** and in it, we can use **variables**, which are created with **$1**, and in place of the unit can be placed other text. The example uses a variable: **"console.log('$1');"**.

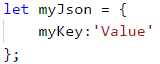
We can use **Tabstops**, which simply **place the cursor in certain places in the code**, and between them can be navigated with tabs. They are created automatically by creating a variable. We can also use **Placeholders**, they are a type of **Tabstops**, but they can also **contain some value**, for example: **${1:myVal}**.

There are also more complex configurations, but for begging these will do an excellent job.

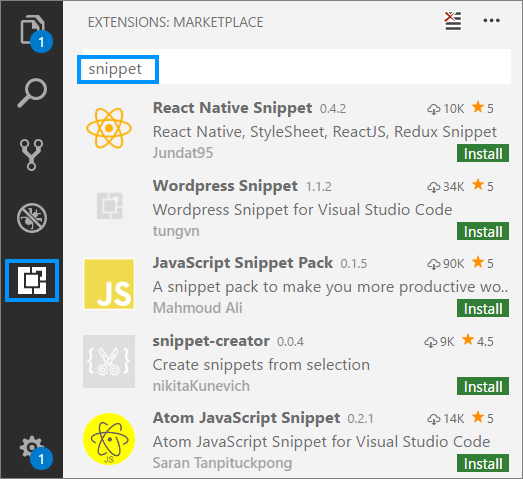
The last part of the template is a **description**, which serves to add an explanation of what it does. Now let's try to make our template. We delete the given example and enter the following code:



Now when we type **json** + [**Tab**] + [**Tab**] in an open JavaScript file in Visual Studio Code, **our new snippet** appears:



For those of you who are more interested in the topic, many of the big **frameworks** like **Angular, React, etc.** have their own templates that can be installed from the **Extensions** window. As the name suggests, a **framework** is a conceptual structure that helps us by giving us some things in advance, and also prevents us from making big mistakes by imposing certain restrictions. The main idea is to provide a complete solution in a given area, which can upgrade all components in this area. Some of the things you can get ready-made are these templates.



## Code Debugging Techniques

Debugging plays an important role in the process of creating software, which is to allow us to **follow the implementation of our program step by step**. With this technique, we can **follow the values of the local variables** because they are changing during the execution of the program and **remove** possible **errors** (bugs). The process of debugging includes:

* **Finding** the problems (bugs).
* **Locating** the code, which causes the problems.
* **Correcting** the code, which causes the problems so that the program works correctly.
* **Testing**, to make sure that the program works correctly after the corrections we have made.

**Visual Studio Code** gives us **a built-in debugger**, thanks to which we can place **breakpoints**, at places we have chosen. When it reaches a **breakpoint**, the program **stops running** and allows **step-by-step running** of the remaining lines. Debugging allows us to **get the details of the program** and see where exactly the errors occur and what is the reason for this.

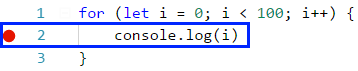
To demonstrate how to use the debugger, we will use the following program:

for (let i = 0; i < 100; i++) {

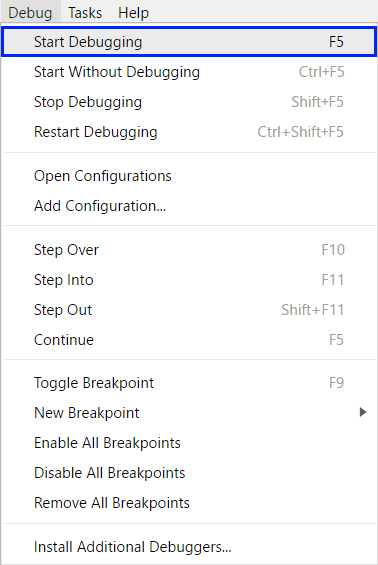
console.log(i);

}

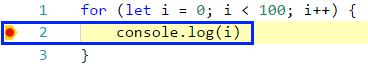
We will place a **breakpoint** on the method **console.log(…)**. For this, we will need to move our cursor to the line, which prints on the console, and press [**F9**]. A **breakpoint** (the red point, just before the number on line 3), where the program will **stop** its execution:



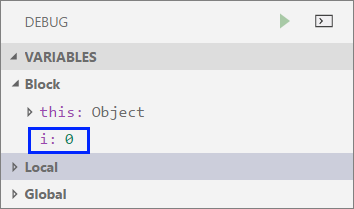
To start **the program in debug mode**, we choose [**Debug**] -> [**Start Debugging**] or press [**F5**]:



After starting the program, we can see that it **stops executing** at line 4, where we placed our breakpoint. The code in the current line **is colored in yellow** and we can **run it step by step**. To go to **the next line** e use the key [**F10**]. **We can see that the code on the current line hasn't been executed yet**. **It will execute when we go ahead to the next line**:



From the **Debug** window, which opens with [**View -> Debug**] or with the keyboard shortcut [**Ctrl + Shift + D**], we can observe **the changes in the local variables**.



## A Guide for Tricks

In this section, we will briefly show **tricks and techniques** of programming with the **JavaScript** language, some of which are already mentioned in this book, which will be very useful, if you take a programming exam for beginners.

### Rounding Numbers

In case we need to round numbers we can use one of the following methods:

* **Math.round(…)** - accepts 1 parameter - **number, which we want to round**. Rounding is performed according to the basic rule of rounding - if the decimal part is less than 5, rounding is down and vice versa, if it is greater than 5 - up:

let number = 5.439;

console.log(Math.round(number));

// This will print on the console "5"

let secondNumber = 5.539;

console.log(Math.round(secondNumber));

// This will print on the console "6"

* **Math.floor(…)** - in case we want the rounding always to be **down to the previous integer**. For example, if we have the number 5.99 and use **Math.floor (5.99)** , we get the number **5**:

let numberToFloor = 5.99;

console.log(Math.floor(numberToFloor));

// This will print on the console 5

* **Math.ceil(…)** - in case we want the rounding always to be **up to the previous integer**. For example, if we have the number 5.13 and use **Math.ceil(5.13)**, we
* will get the number **6**:

let numberToCeil = 5.13;

console.log(Math.floor(numberToCeil));

// This will print on the console 6

* **Math.trunc(…)** - in case we want to **remove the fractional part**. For example, if we have the number 2.63 and use **Math.trunc(2.63)**, we will get the number **2**:

let numberToTrunc = 2.63;

console.log(Math.floor(numberToTrunc));

// This will print on the console 2

### Use === Instead of ==, and !== Insted of !=

The operators **==** and **!=** make **automatic conversion** of the compared value or variable, while the operators **===** and **!==** does not make such a conversion and if both values are not of the same type - the result is **false**. They (**==** and **!=**) make a comparison of **value and type**, which is more accurate and even faster. Let's look at the following example to clarify what is meant by data **type**:

[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

We see how the number **10** can be written in our programs in different ways. Written in this way **[10]** is an **array** of one number. In short, arrays are **multiple values** stored to a variable. For example:

let array = [10, 20, 30, 40];

// This is a variable of type array

We will learn more about arrays later but for now, let's just think about whether the array **[10]** is normally equal to the number **10**. We will give you a hint - **it is not normal**. Therefore, if we do not want unpleasant errors (bugs) in our programs, it is best to use the operators **===** and **!==**.

The situation with other comparison operators is similar and the same logic applies there.

### How to Write a Conditional Statement?

The conditional **if statement** contains the following elements:

* Keyword **if**
* **A Boolean expression** (condition)
* **Body** of the conditional construction
* Optional: **else clause**

if (condition) {

// body

}

else (condition) {

// body

}

To make it easier we can use a code snippet for an **if construction**: **if** + [**Tab**] + [**Tab**].

### How to Write a 'For' Loop?

For a **for loop** we need a couple of things:

* Initializing block, in which the counter variable is declared (**let i**) and its initial value is set.
* Condition for repetition (**i <= 10**).
* Updating statement (**i++**).
* Body of the loop.

for (let i = 0; i <= 10; i++;) {

// body

}

To be easier to write it, we can use the code snippet for the **for loop**: **for** + [**Tab**] + [**Tab**].

### Using The So-Called Positive (Truthy) and Negative (Falsy) Value

All **Truthy** values used in the **if** conditional statement will give a positive result and accordingly or program will continue ist execution in the body of the conditional statement (for the example here the body of the conditional statement is not formatted correctly).

For some of them it seems logical to give a positive result, but for others not so much.

//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

Conversely, all **Falsy** values will give a negative result and the program will not enter the body of the conditional statement.

//Falsy

if (false) {} //false

if (null) {} //false

if (undefined) {} //false

if (0) {} //false

if (NaN) {} //false

if ('') {} //false

if ("") {} //false

It is not necessary to know these values by heart at this stage, but only to remember that there are so-called **Truthy** and **Falsy** values. Over time, we will get used to how to use them properly and how they help us shorten our code.

## What Have We Learned from This Chapter?

In the current chapter, we learned how to **correctly** **format** and **name** the elements of our code, some **shortcuts** for work in Visual Studio Code, some **code snippets**, and how to **debug code**.

# Conclusion

If you have **read the entire** book and you've solved all the problems from the exercises and reached the present conclusion, **congratulations**! You've already made the **first step** in learning the **profession of a programmer**, but there is a **long way** to go until you become **really good** and make **software writing** your **profession**.

Remember the [**four main groups of skills**](https://js-book.softuni.org/chapter-00.2-how-to-become-a-programmer.html) that each programmer must have so we can work in the industry:

* Skill #1 – **writing the program code** (20% of programmer's skills) – covered to a large degree by this book, but you must learn additional basic data structures, classes, functions, strings, and other elements of code writing.
* Skill #2 – **algorithmic thinking** (30% of programmer's skills) – covered partially by this book and developed mostly by solving a large amount of diverse algorithmic problems.
* Skill #3 – a **fundamental understanding of the profession** (25% of programmer's skills) – acquired for a few years in combination with learning and practice (reading books, watching video lessons, attending courses, and mostly by writing diverse projects in various technological areas).
* Skill #4 – **programming languages and software technologies** (25% of programmer's skills) – acquired in a long period, by a lot of practice, consistent reading, and writing projects. Such knowledge and skills quickly get outdated and need to be updated frequently. Good programmers are involved in studying new technologies every day.

## This Book is Only The First Step!

**The present** book on programming basics is just the **first step** in building the skills of a programmer. If you were able to solve **all problems**, this means you have **obtained valuable knowledge** in the programming principles with **JavaScript** language on a **basic level**. You are about to start **in-depth** studying of programming, develop **your algorithmic thinking**, and then add **technological knowledge** regarding JavaScript language, Node.js ecosystem (Node.js, npm, Express.js, etc.), front-end technologies (HTML, CSS, Angular, React, AJAX, HTML5) and many other concepts, technologies and instruments for software development.

If you were **not able** to solve all problems or a large part of them, go back and solve them! Remember that **becoming a programmer** requires **a lot of work and effort**. This profession is not for lazy people. There is no way to learn it unless **you seriously practice** programming for years!

As we already explained, the first and basic skill of a programmer is **to learn to write code** with ease and pleasure. This is namely the mission of this book: to teach you how to code.

## What to Do After Reading This Book?

This book **gives you solid foundations**, thanks to which it will be easy for you to continue developing as a programmer. If you wonder how to continue your development, you have the following possibilities:

* to study for a [**software developer**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) **at SoftUni** and make programming your profession.
* to continue developing as a programmer **on your own**, for example through self-training or via free online lessons.
* to **stay at a coder level**, without going more seriously into programming.

### Study Software Engineering at SoftUni

The first option, which we recommend, is to enroll in a [**Software Developer program**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) to master programming on a professional level. The SoftUni curriculum is carefully developed by **Dr. Svetlin Nakov and his team**, to provide you consequently and with gradually increasing complexity all the skills that a software engineer must have, **to start a career as a software developer** in an IT company.

SoftUni’s self-paced programs are designed as a **sequence of modules** giving you fundamental knowledge in software development and practical skills to prepare you to work as a programmer with the latest software technologies. The classes are divided into **learning theory (30%)** and doing exercises and **projects (70%)**.

### It Takes at Least a Year of Intense Code Writing to Become a Programmer

Keep in mind that **to become a programmer takes a lot of effort**, writing tens of thousands of lines of code, and solving hundreds, even thousands of practical problems, and this takes years! If someone offers you "**an easier program**" and promises you to become a programmer and start working within 3-4 months, then either they are **lying** to you, or they will give you such a low level, that **companies won't even take you as a trainee**, even if you pay to the company that is wasting its time with you. There are exceptions, of course, for example, if you are not starting from scratch, or if you have extremely well-developed engineering thinking, or if you apply for a very low position (for example technical support), but in general, **you cannot become a programmer if you haven't spent at least 1 year of intense learning and code writing**!

### How Many Hours Per Day Does The Training Take?

The training for software engineers at SoftUni is a **very serious occupation** and you need to spend **at least 4-5 hours every day**, preferably your entire attention and time. Combining **training with other obligations** is not always easy. Of course, it is always best if you can dedicate all your time and attention to mastering programming but SoftUni’s **self-paced** [**Software Developer**](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) programs make it possible to learn at your own pace. They are great for high school pupils, university students, and people with full or part-time jobs. And remember, you cannot become a successful software developer by only practicing 2-4 hours a week. As a minimum, you need to spend at least 20-30 hours a week learning to code.

### SoftUni for People Who Work and Study

We recommend to everyone who gets **an excellent score at Judge Problems** and is passionate about making programming their profession, to consider leaving the rest of their commitments aside to **spend more time** on learning the profession of a software engineer and start making a living through it.

We recommend to all who **cannot get an excellent score at the Judge problems** to spend more time on better learning, understanding, and most of all, practicing the material studied in the present book. If you cannot easily solve the problems in this book, you will not be able to cope with programming and software development in the future.

### Study Software Engineering on Your Own

Another possibility to develop after this book is to **continue studying programming outside of SoftUni**. You can enroll or subscribe to **video training**, that goes into more details in programming with **Python** or other languages and development platforms. You can **read books** on programming and software technologies, follow **online tutorials** and other online resources – there are plenty of free materials on the Internet. However, keep in mind that the most important thing about the profession of a programmer is **to do practical projects**!

**You cannot become a programmer without a lot of code writing and intense practice**. Allocate **sufficient time** to it. You cannot become a programmer for a month or two. On the Internet, you will find a wide variety of **free resources** such as books, manuals, and video lessons. However, you need to invest **at least a year or two**, to acquire a foundation level, needed for starting a job.

After you gain some experience, find a way to start **an internship in a company** (which will be almost impossible unless you'd spent at least a year of intense code writing before that) or come up with **your practical project**, on which you need to spend a few months, even a year, to learn based on the trial-and-error principle.

|  |  |
| --- | --- |
|  | Keep in mind that there are many ways to become a programmer, but they all have something in common: **code writing and years of practice**! |

## Online Communities for Beginners in Programming

Regardless of the path you have chosen, if you are seriously involved in programming, we recommend subscribing to specialized **online forums, discussion groups, and communities**, from which you can get assistance from your colleagues and track the novelties in the software industry.

If you aim to study programming seriously, **surround yourselves with people who are involved in programming** seriously. Join **communities of software developers**, attend software conferences, go to events for programmers, find friends with whom you can talk about programming and discuss problems and bugs, find an environment that can help you. In most large towns there are free events for programmers, a few times a week. In smaller localities, you have the Internet and access to the entire online community.

Here are some recommended **resources**, that will be useful for your development as a programmer:

* [https://learn.softuni.org](https://learn.softuni.org/catalog?utm_source=JavaScript&utm_medium=text&utm_campaign=free+programming+books) – the official page for SoftUni’s **self-paced Software Developer programs and open courses**.
* <https://www.facebook.com/softuni.org> – official **Facebook page of SoftUni**. By it, you will learn about new courses, seminars, and events related to programming and software development.
* [https://introprogramming.info](https://introprogramming.info/) - official website of the **books "Programming Basics"** with **C#**, **Java**, **JavaScript**, and **Python** by Dr. Svetlin Nakov and his team. The books examine in-depth programming basics, basic data structures and algorithms, object-oriented programming, and other basic skills, and are an excellent continuation for reading after the current book. However, **besides reading, you need to do intense code writing**, do not forget that!
* [https://stackoverflow.com](https://stackoverflow.com/) - **Stack Overflow** is one of the **widest** discussion forums for programmers worldwide, where you will find assistance for any possible question in the world of programming. If you are fluent in English, look up at StackOverflow and ask your questions there.
* [https://www.meetup.com](https://www.meetup.com/) - search **tech meetups** around your town and enter the community that you like. More of the tech meetups are free and the newcomers are welcome.

## Good Luck to All of You!

On behalf of the entire authors' team, we **wish you endless success in the profession and your life**! We will be really happy if we have helped you get **passionate about programming** and we have inspired you to go bravely towards becoming a "software engineer", which will bring you a good job that you will work with pleasure, give you a high-quality life, and prosperity, as well as amazing perspectives for development and possibilities for making impressive projects with inspiration and passion.

*Sofia, May 30, 2019*