# Guidelines for Creating Course Exams

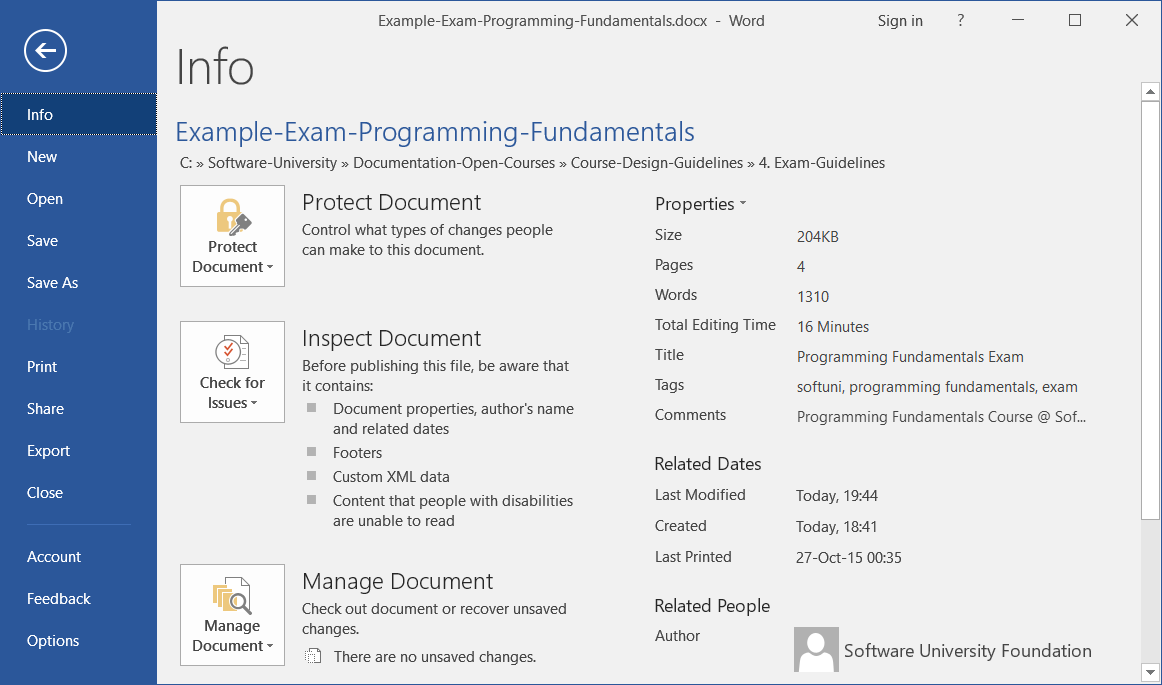
This document describes the **guidelines** for creating course **exam** assignments for the internal training program at the Software University (SoftUni).

## Objectives

* **Match the course exam planning**
  + Problems at the exam should **match the topics** covered in the course with focus on the most important learning objectives.
    - **Problem #1** – covers the **first 2-3 topics** of the course.
    - **Problem #2** – covers the **next 2-3 topics** of the course.
    - …
    - **Last problem** – covers the last few topics (except the optional / bonus topics).
  + **Example** - Technology Fundamentals Mid Exam Planning:
    - **Problem 1: Conditional Statements and Loops**
      * Knowledge and Skills to evaluate
        + Comparison operators
        + Logical Operators
        + If-else Statement
        + Switch-Case Statement
        + Loops
        + String Concatenation
        + Formatting
    - **Problem 2:** 
      * **…**
* **Short and clear descriptions**
  + **Describe** the problems well, but in **short form**, without unneeded details or stories.
  + Use **examples** instead of long text descriptions.
  + Use **pictures** / **diagrams** / **screenshots** to illustrate the problem visually.
  + Provide well prepared **sample input and output** + description of the sample (when it is not obvious).
* **Increasing difficulty**
  + The **first problem** should be **the easiest**. It is designed to be **solved by almost all students**. It should cover the **first 2-3 topics** and should be **at the level of the first 2-3 exercise problems** for the covered topics. It should include no tricks, special cases or advanced techniques.
  + The **next few problems** should cover the **next few topics**. Their difficulty should be like **the average in the exercises** for the covered topics. These problems may include non-straightforward techniques shown in the lessons and exercises.
  + The **last problem** may be **a bit more difficult**. It should be designed for students above the average level, aiming to get an excellent grade. It is not obligatory to have **more complex** or **larger** problem, but it is OK to have it.
* **Automated check-up (with a judge system)**
  + Prefer problems designed to be **automatically evaluated** using a judge system.
  + Think about the **check-up**: either use the judge system or provide **well defined evaluation criteria** (e.g. score for each problem / sub-problem).

## Document Structure

* Document **metadata** – ensure your document holds correct metadata (title, tags, comments, author):



* + **Title**: ***{course name}*** Exam
  + **Tags**: ***{course tag 1}***, ***{course tag 2}***, ***{course tag 3}***, …, Software University, SoftUni, programming, coding, software development, education, training, course
  + **Comments**: ***{course name}*** Course @ SoftUni - <https://softuni.bg/courses/>... (course URL)
  + **Categories**: programming, education, software engineering, software development
  + **Subject**: ***{course name}***
  + **Author**:Software University Foundation

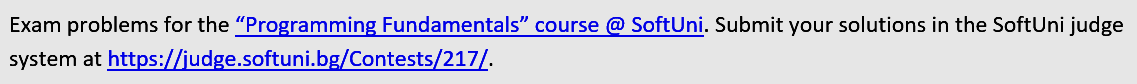
Maintain **consistent metadata** for the entire course (similar metadata for all course documents).

* Document **title** – should match the **course name** + **exam date**, e.g.



* Document **intro section**
  + Explains the **purpose of the document**: exam for certain course (+link) + link to submit the solutions (in the judge).

Example:



* **Problems** (all problems follow the same structure)
  + Short **title** – explain in 1-3 words the problem (like the subject in an email)
  + Short **description** – keep it short, explain by examples and figures, not with long text
  + **Input** and **output** description – describe shortly the input and output format (if available)
  + **Constraints** – only when appropriate, as addition to the input / output description
  + **Examples** / **screenshots** – obligatory put examples: sample input, sample output, figure (when appropriate), screenshot (when appropriate)
  + **Hints (optionally)**: Some problems may provide hints to give some directions to the students.
* Problems should **follow the course topics**: earlier problems should correspond to earlier topics.
* I the same time, order the problems in **increasing complexity**: from the easiest to the hardest!

## How to Describe Problems?

### Problem Title

Use **short title** that answers the question “**what is this problem about?**”.

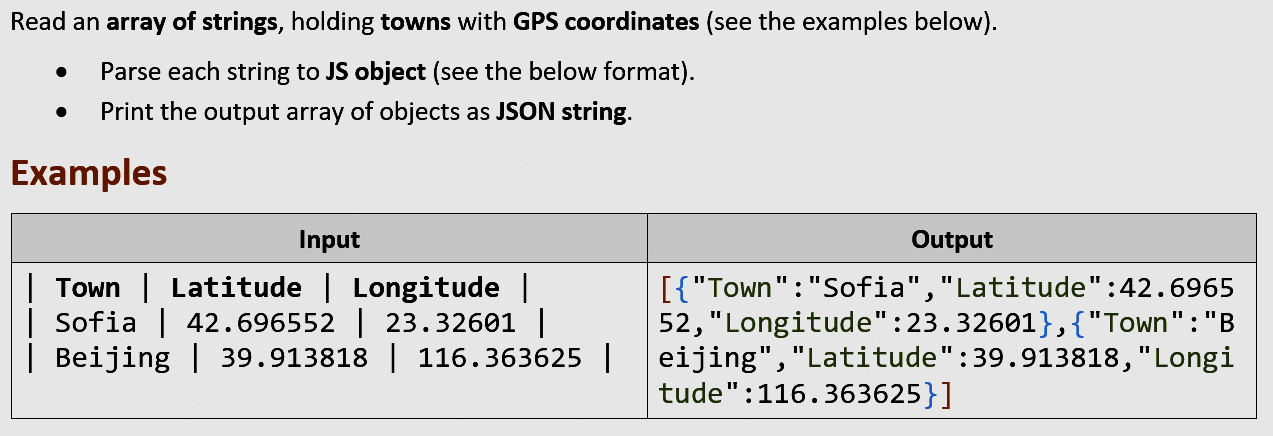
* Examples of **good names**: “Point in Rectangle”, “Age Calculator”, “Biggest of 3 Numbers”, “Voting System”.
* Examples of **bad names** (titles too long and descriptive):“Calculate the Biggest Number from Array of Integers”, “Read Numbers, Sort then and Print them in JSON Format”, “HTML Page Holding Students”.
* Examples of **bad names** (titles too short and say nothing about the problem): “Biggest”, “Number”, “Age”.

### Problem Description

Describe the problem **shortly**, without unneeded explanations and without meaningless stories.

Use **examples** and **figures** instead of long text descriptions. Don’t describe the input and output format when it is visible from the examples (unless something non-obvious should be clarified).

* **Good example** of short and clear description with a colored input / output:

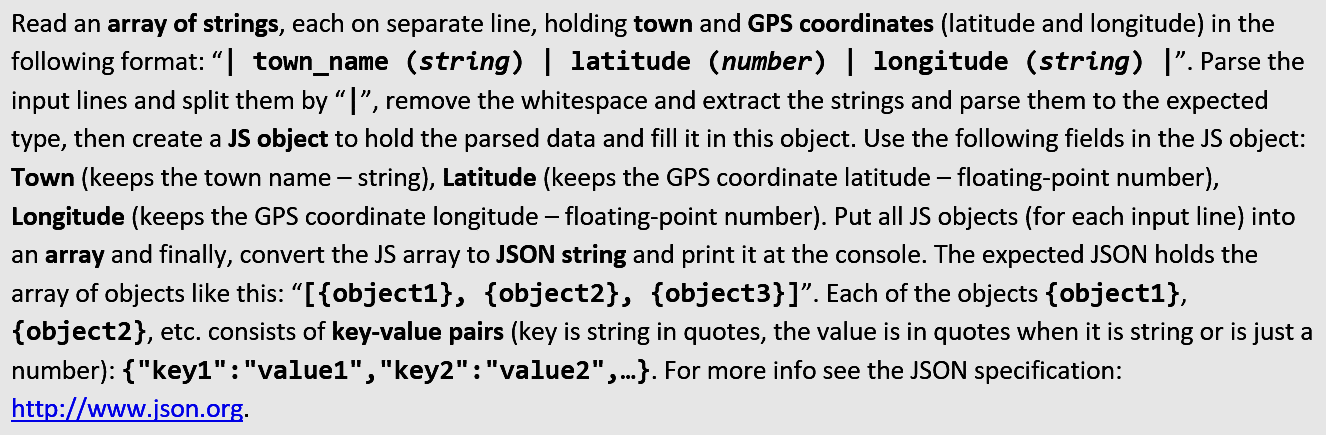


The above description is very **short** and still **understandable**. It is explained very well through **examples**.

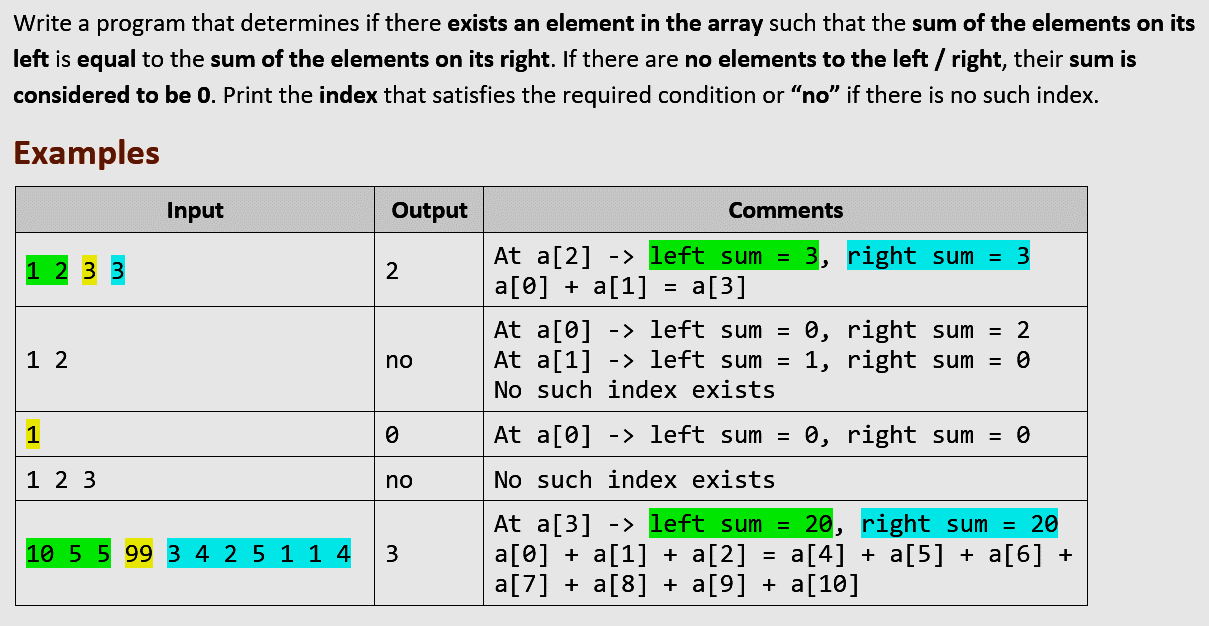
To be more precise, you should describe briefly the **input** and **output format**. This is not obligatory, when the input and output formats are clearly visible from the examples but is recommended.

Also, you should describe the input **constraints** (in case they matter).

* **Bad example** of too descriptive text. This example is a **disaster** for the students, don’t do it! It explains the same problem in over-descriptive way, with long explanations instead of clear examples.



* Another **good example**:



Why the above example is **good**?

* + Thе problem is best described through **examples**, not through text.
  + **Short description** of the problem, just a few sentences.
  + Good **examples**, covering the **normal** and **special cases**.
  + **Colors** to help understanding the examples, focusing on certain portions of the input.
  + **Comments** to help understand each of the examples.

### Input / Output + Constraints

* Keep **short** the input and output format descriptions. **Examples** will explain them better than a text.
* Explain the **non-obvious information**.
  + For example, if you enter 4 numbers, describing rectangle location + size, it is critical to explain, that the input holds at each line **left**, **top**, **width** and **height**, not just 4 numbers.
* **Constraints** should be provided for most problems but **give good examples** instead of long descriptions.
  + For example, instead of explaining that the rectangle location and size could use **floating-point numbers**, just give examples holding non-integer values.
  + **Constraints** should explain non-obvious limitations, like “**0 ≤ hours < 12**” and “**0 ≤ minutes < 60**”.
  + If the constraints are not important or are absolutely **obvious**, you skip them. For example, if the problems says “check a number and print whether it is ‘odd’, ‘even’ or ‘non-integer’”, no constraints may be described. Just **give good examples** (positive number, negative number, 0, non-integer number, non-number string, etc.).
* A **good example** of input / output / constraints section from the problem description:



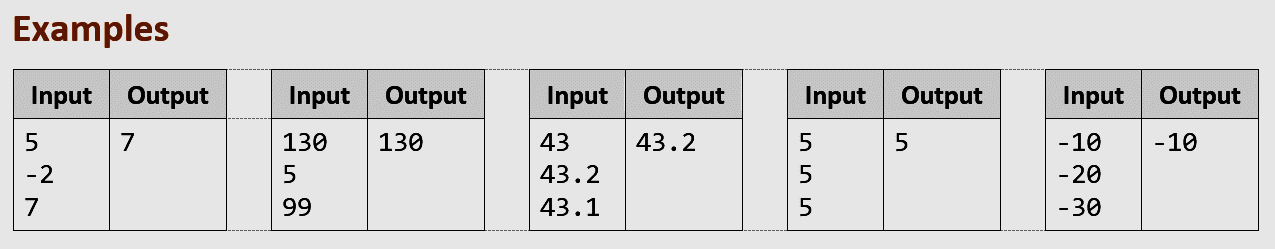
### Examples (Sample Input / Output)

**Examples** (sample input + output) are **very, very, very important**, even the most important part of each problem description!

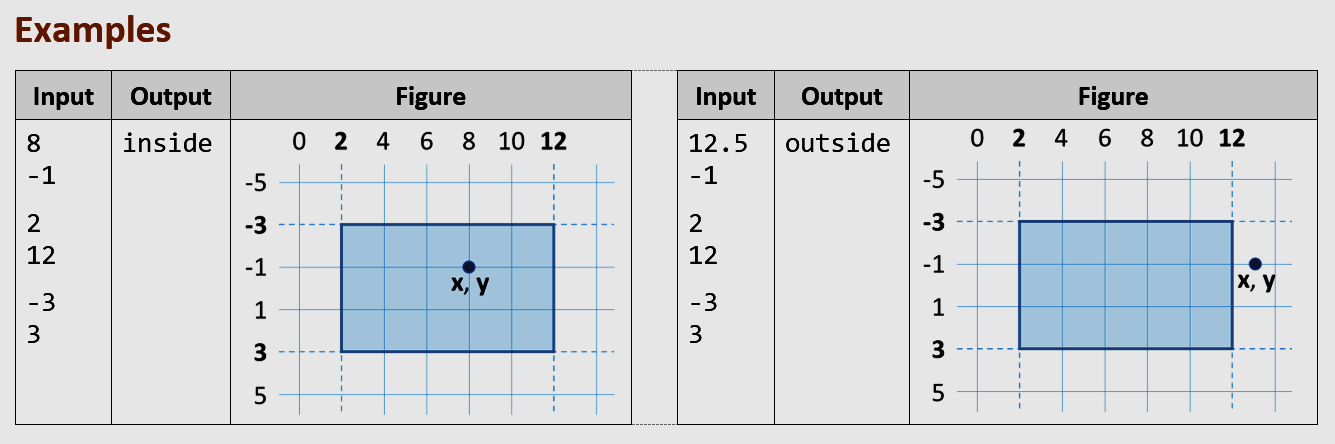
* Give **good examples**! Good examples cover the **normal case** (usual case) and all **special cases** (border cases).
  + A **big mistake** is to have a **special case, not covered in the examples**.
  + Examples should be **the main source of information** about the problem, its allowed input and its expected output.
* Give **several examples**, not just 1 or 2. Complex problems may give 5-10 examples.
* Instead of explaining what to print in a **special situation**, just **give an example** to make it obvious.
* Use **colors** or **figures** or **comments** to explain each test case (when needed).
* For **UI problems** (e.g. in the HTML course) provide **screenshots**, which make obvious the requirements.

Examples of **good sample input + output**:

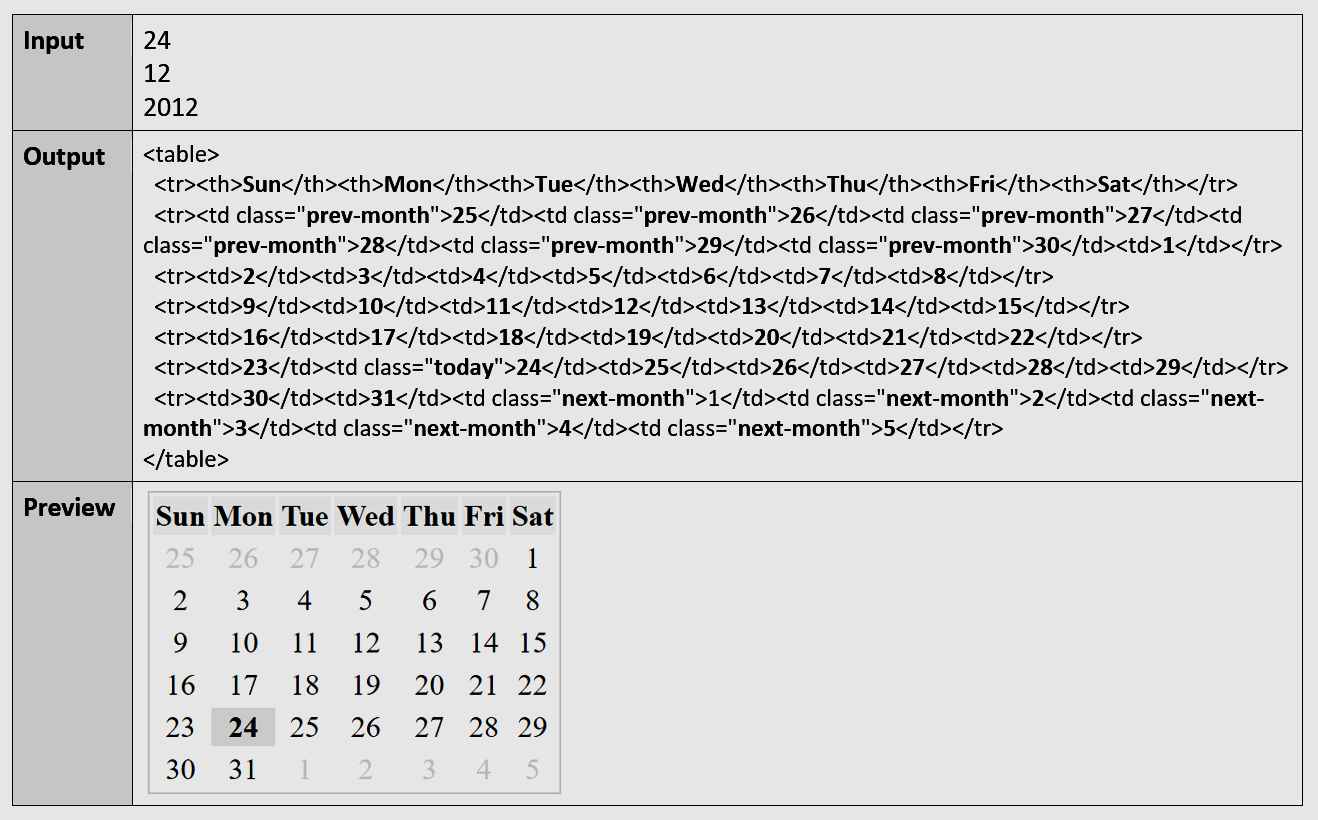
* Assume the problems says “*Find the* ***biggest 3 numbers*** *from array of numbers*”. Examples should answer what to do when all numbers are **equal**, what to do with **negative numbers**, what to do with **non-integers**.
  + The below examples **implicitly say** that the expected input consists of **3 lines** and each line holds a **number**.
  + If the problem allows **non-integers**, you should obligatory put such an example.
  + If the problem allows **less than 3 numbers**, you should obligatory have such an example.
  + If the problem allows **more than 3 numbers**, you should obligatory have such an example.
  + From the example below, it is clear that the input consists of **3 lines**, each holding a **real number**:



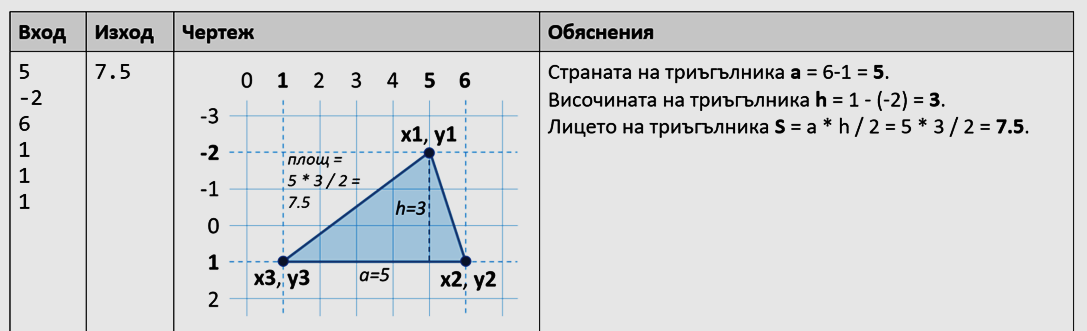
* Input which is **meaningless unless visualized**  just visualize it. Examples of such problems might be geometry (points and figures in the plane), transformations of elements that need visual explanation or UI-related problems that need a screenshot.



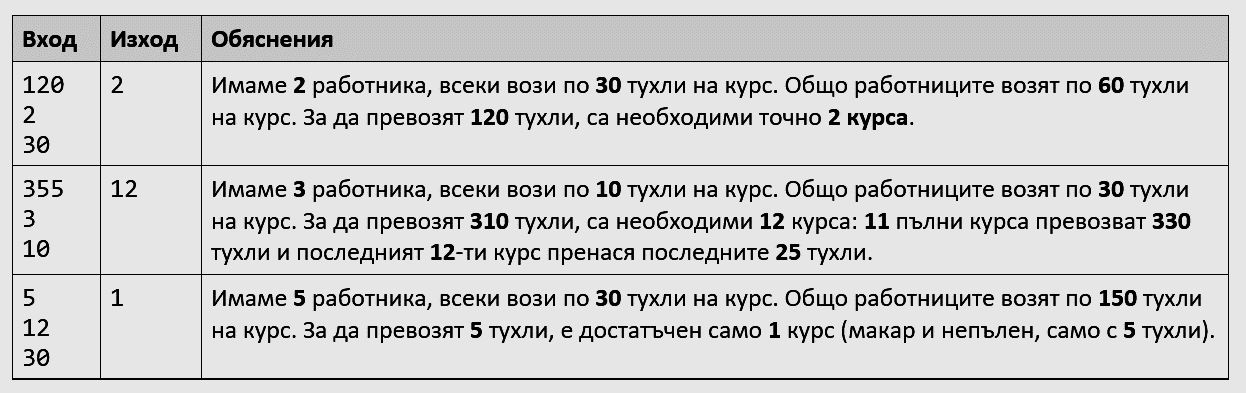
* Output, which is **hard to be perceived** unless visualized. The example below shows how important is to **visualize** certain input or output:



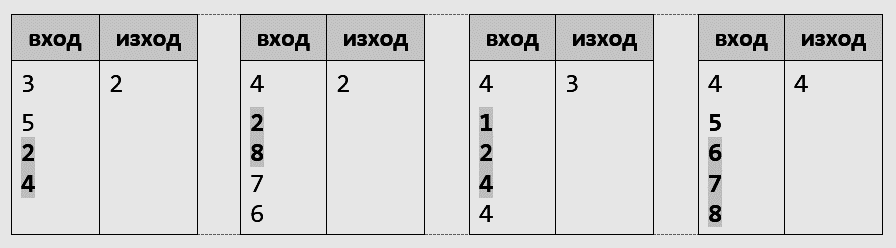
* Example of **visualization** + **explanations** for each example:



* Example of good **example explanation**:



* Example of using **colors / shading** to **improve the examples**:



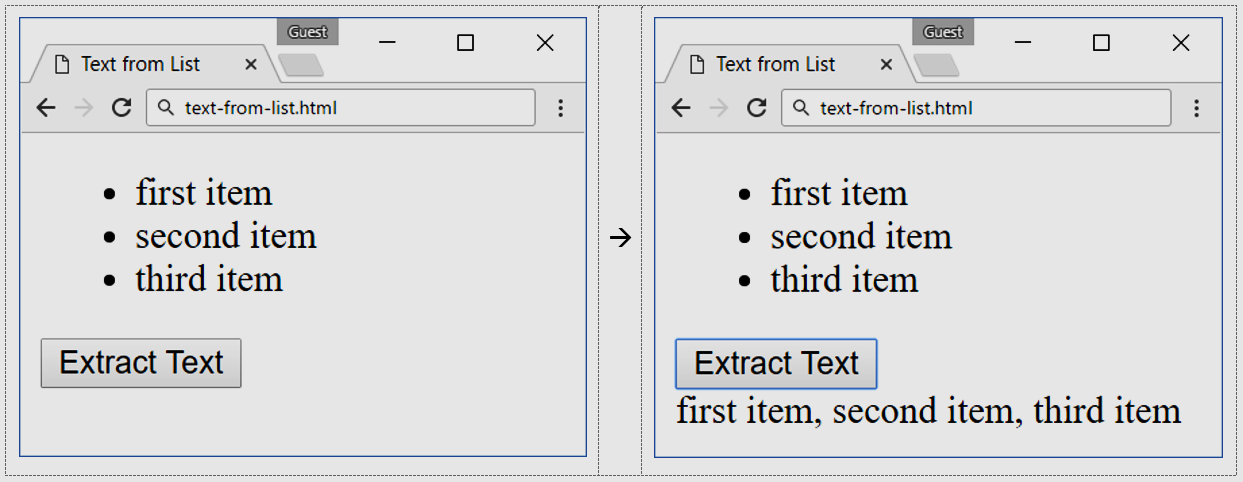
### Screenshots

For problems, where **UI** (user interface) is involved, the **screenshots** are **very, very, very important**, even the most important part of each problem description! Most UI problems need just a **few screenshots** and **very little text**.

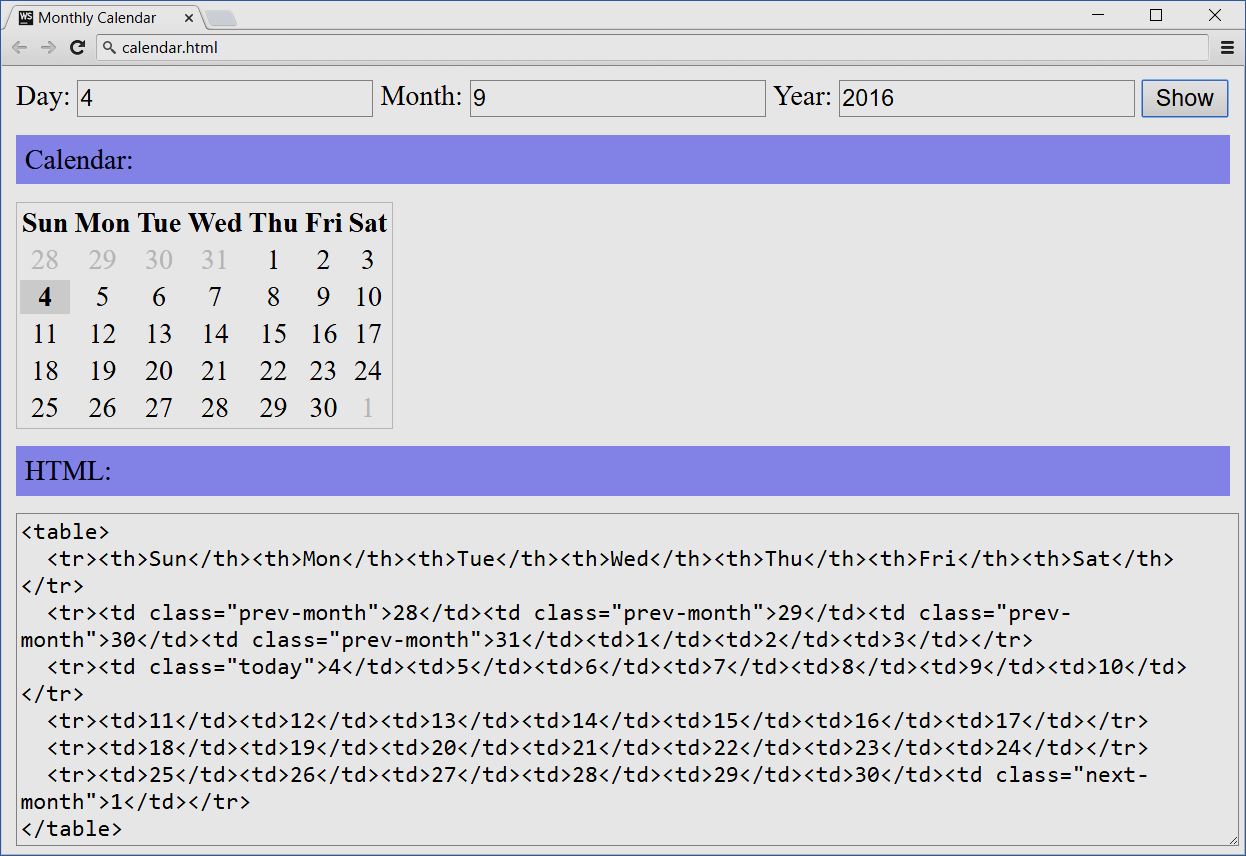
Use **good screenshots** to **explain visually the application functionality**. Good screenshots are **small images** using **large fronts**, showing the interesting part of the screen only (without unneeded areas) and showing what happens during the **user interaction** with the application (what happens when users click at each link / button / form).

Examples of **good screenshots**, explaining the problem without need of text:

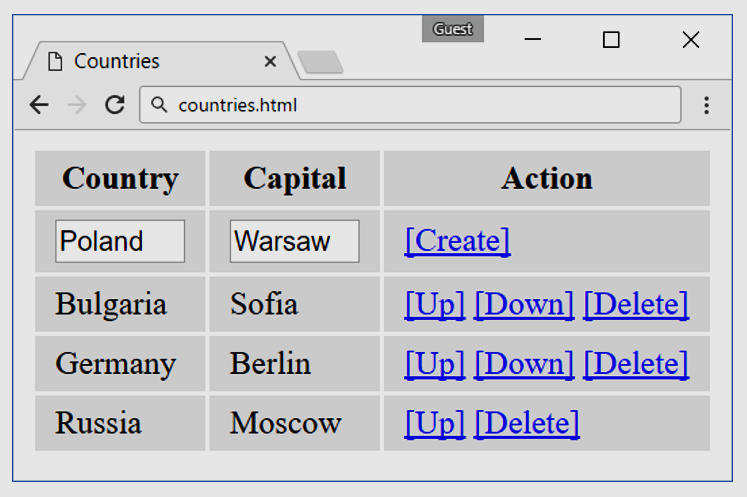
* **Screenshots** showing the **application in action** (what happens on button click in this case):



* The problem in the below example is to **print an HTML calendar** by **day** + **month** + **year**. This screenshot explains everything in big details and no explanatory text is needed:



* This **screenshot** **explains the problem** at 80%. Even if you say “just create this app”, it is clear what is expected. Most UI problems need just a **few screenshots** and **very little text**.



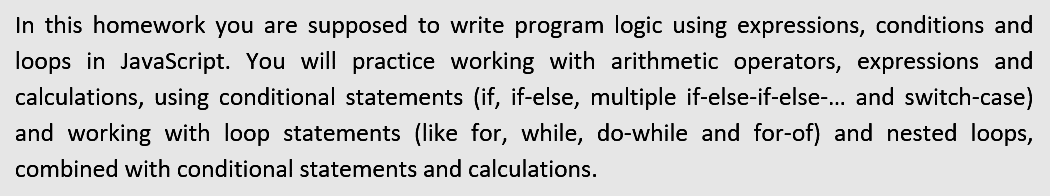
## Document Formatting

Use the **document template** from the sample exam documents. They hold the predefined styles for all elements:

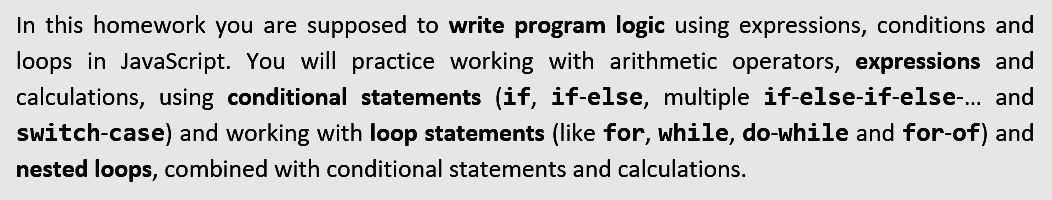
* **Heading 1** style – for the document title.
* **Heading 2** style – for the problem titles.
* **Heading 3** style – for the sections inside the problem (e.g. Input / Output, Examples, Hints).
* **Heading 4** style – for sub-sections (if needed).
* **Code** style – for source code elements (like variable names and file names).

### Text Formatting

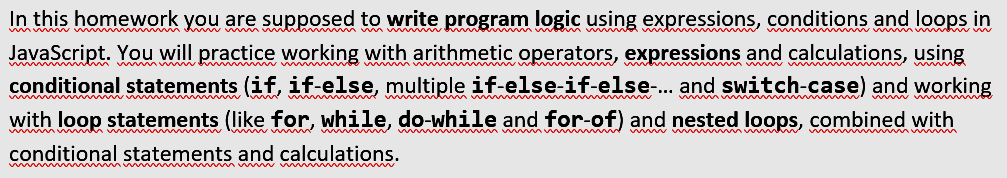
* **Bold the key words / phrases** to enhance text readability. Don’t bold too much, just the most important key words or phrases, at least one per paragraph and at most one per sentence.
* Use the style “Code” from the MS Word styles for pieces for code pieces, e.g. file names, variable names, function names, class names, language keywords, etc.
* **Bad example** (hard to read text with no bold):



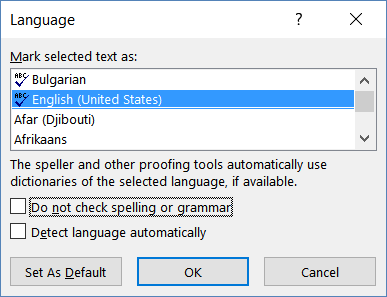
* **Good example** (better formatted text with **bold** for key phrases and Code for identifiers / code elements):



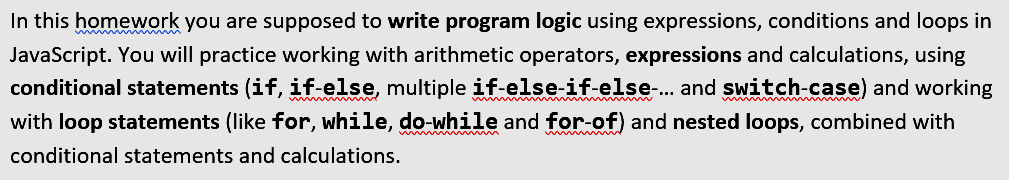
* Ensure the **text spelling and grammar correction** is used properly. **Bad example**:



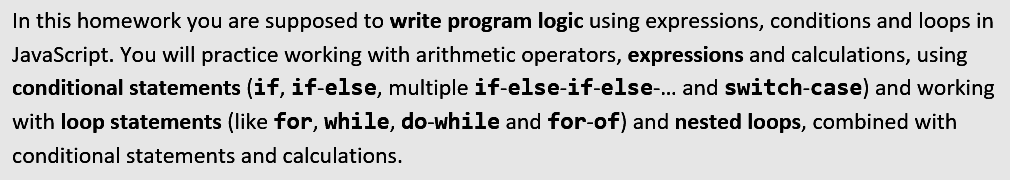
* The above can be fixed by choosing “**English (United States)**” as spelling language + enable spelling:



* After enabling the **spell checker**, it still **may not recognize** (and keep undelined) some pieces of the text, e.g. variable names, file names, etc. **Bad example**:



* You can **disable the spell checker** for these words / phrases checking the **[Do not check spelling or grammar]** in the above dialog box. **Good example**:



### Formatting Titles in the Document

Type the **titles** in the document, starting each word with **upper letter**. Exception: keep lowercase the conjunctionс, prepositions and articles (like ***a*** and ***the***). **Good example** of correctly formatted title:



**Bad examples** of incorrectly formatted titles:





### Code Formatting

When a piece of code is **part of the problem description**, and the student should use it directly by **copy / paste**, provide this code either as an **external file** (e.g. a ZIP archive with files and resources) or (when it is short enough), **put it directly in the text**. When you put code snippets, **intended for copy/paste**, format them in readable form:

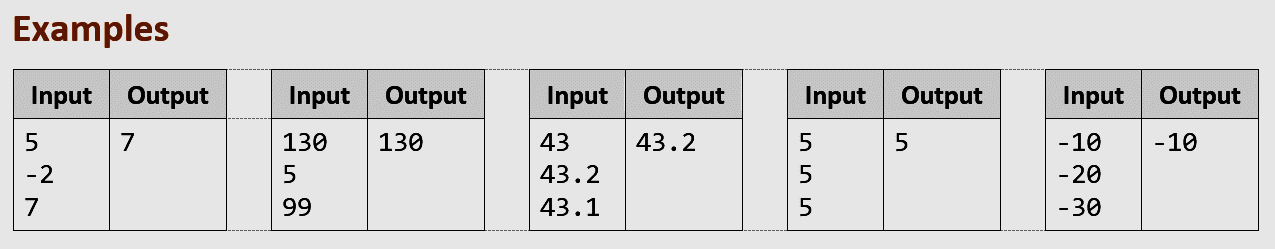
* Copy / paste the **colored code** from your **IDE**. Colors significantly improve code readability.
* Use font Consolas, 11pt. You may copy / paste the code from Visual Studio / Eclipse / Web Storm to preserve its coloring and later change the font in the MS Word document.
* Beware to preserve the correct **indentation of the code** (e.g. 2 or 4 spaces). Sometimes the **[Tab]** character is not displayed correctly in MS Word, so check the indentation after you paste the code.

Example of **well formatted source code**, intended to be copy / pasted:

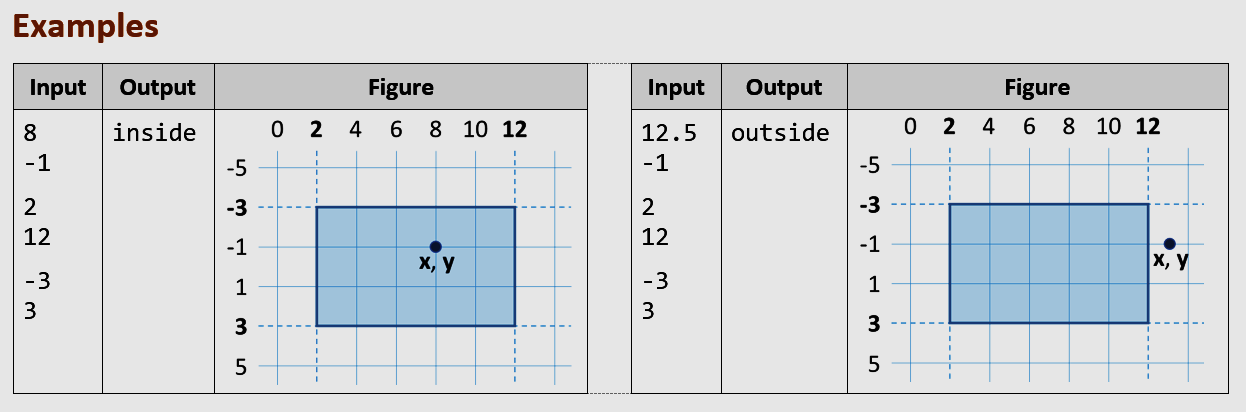
|  |
| --- |
| **calendar.html** |
| <!DOCTYPE html> <**html**> <**head**>  <**title**>Monthly Calendar</**title**>  <**style**>  .**prev-month**, .**next-month** { color: **#CCC** }  .**today** { font-weight: **bold**; background: **#DDD**; }  .**title** { background: **#AAAAFF**; margin: 10**px 0**; padding:5**px** }  **table** { border: 1**px solid #CCC**;}  **td** { text-align: **center**; }  **#calendarCode** { width: 100%; }  </**style**>  <**script**>  **function** *calendar*([day, month, year])  {  *//* ***TODO: return the HTML text holding the calendar table***  }  </**script**> </**head**>  <**body**>  Day: <**input** id=**"day"** type=**"number"** value=**"4"** />  Month: <**input** id=**"month"** type=**"number"** value=**"9"** />  Year: <**input** id=**"year"** type=**"number"** value=**"2016"** />  <**input** type=**"button"** value=**"Show"** onclick=**"let *calendarHTML*** =  *calendar*([**document**.getElementById(**'day'**).**value**,  **document**.getElementById(**'month'**).**value**,  **document**.getElementById(**'year'**).**value**]);  **document**.getElementById(**'calendar'**).**innerHTML** = ***calendarHTML***;  **document**.getElementById(**'calendarCode'**).**innerText** = ***calendarHTML*"** />  <**div** class=**"title"**>Calendar:</**div**>  <**div** id=**"calendar"**>Calendar will be shown here</**div**>  <**div** class=**"title"**>HTML:</**div**>  <**textarea** rows=**"12"** id=**"calendarCode"**></**textarea**> </**body**>  </**html**> |

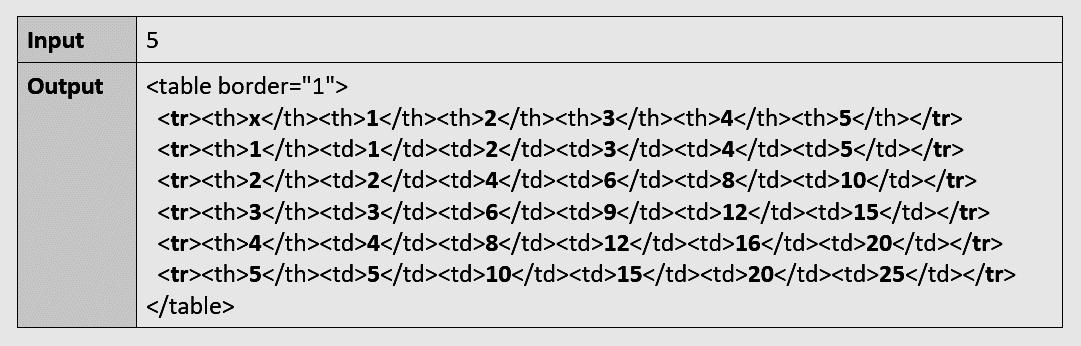
### Input / Output Examples Formatting

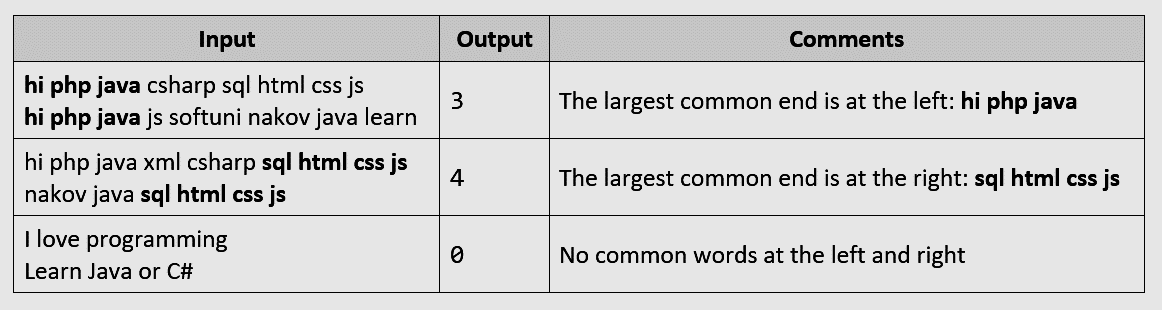
Format the **input / output examples** in **tables** like shown below. When the input / output is short, put several examples in the same table to save space in the document:



When the **input / output is longer**, use different **table layouts**:

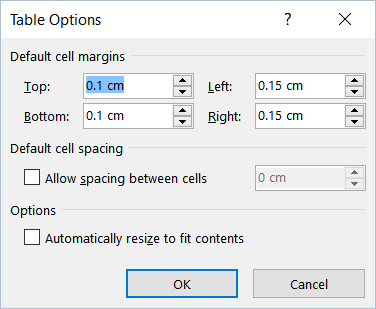
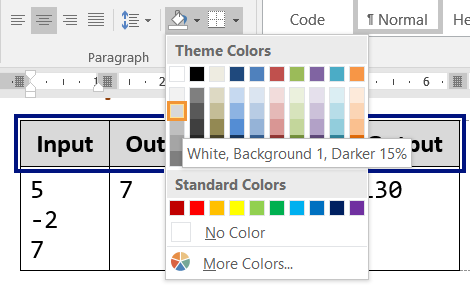


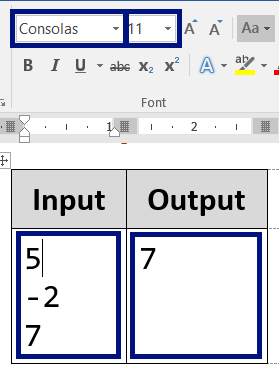
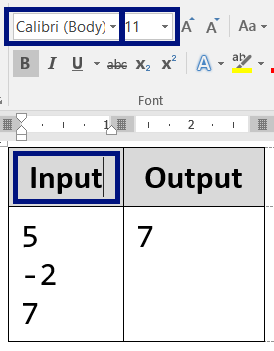
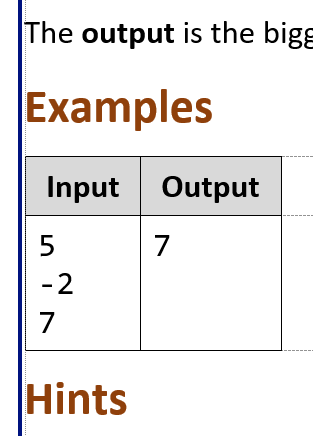




**Format the tables** holding sample input / output like in the sample document:

* **Margins**: 0.1 cm (top + bottom), 0.15 cm (left + right)
* **Header**: background: 15% grey, font: **Calibri, 11pt, bold**
* **Input / output rows**
  + Font: **Consolas, 11pt, bold**
  + You may use a smaller font size by exception to fit the code
* **Left align the table borders** at the same level with the text (see the image below).

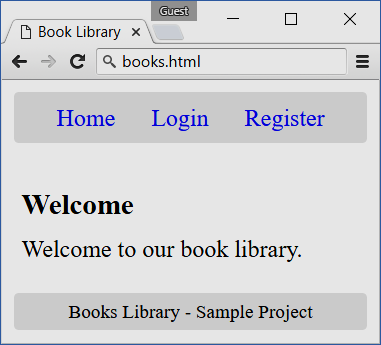
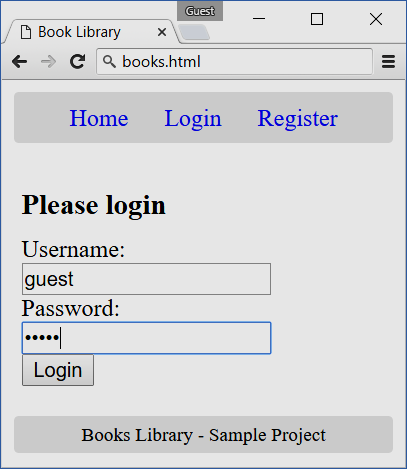
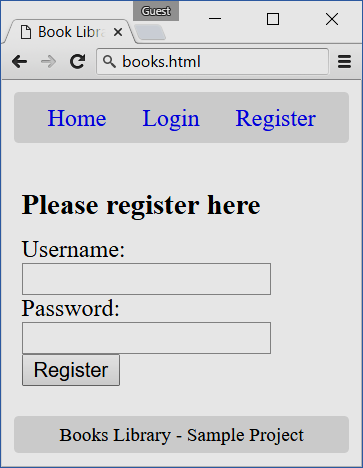
  

### Screenshot Formatting

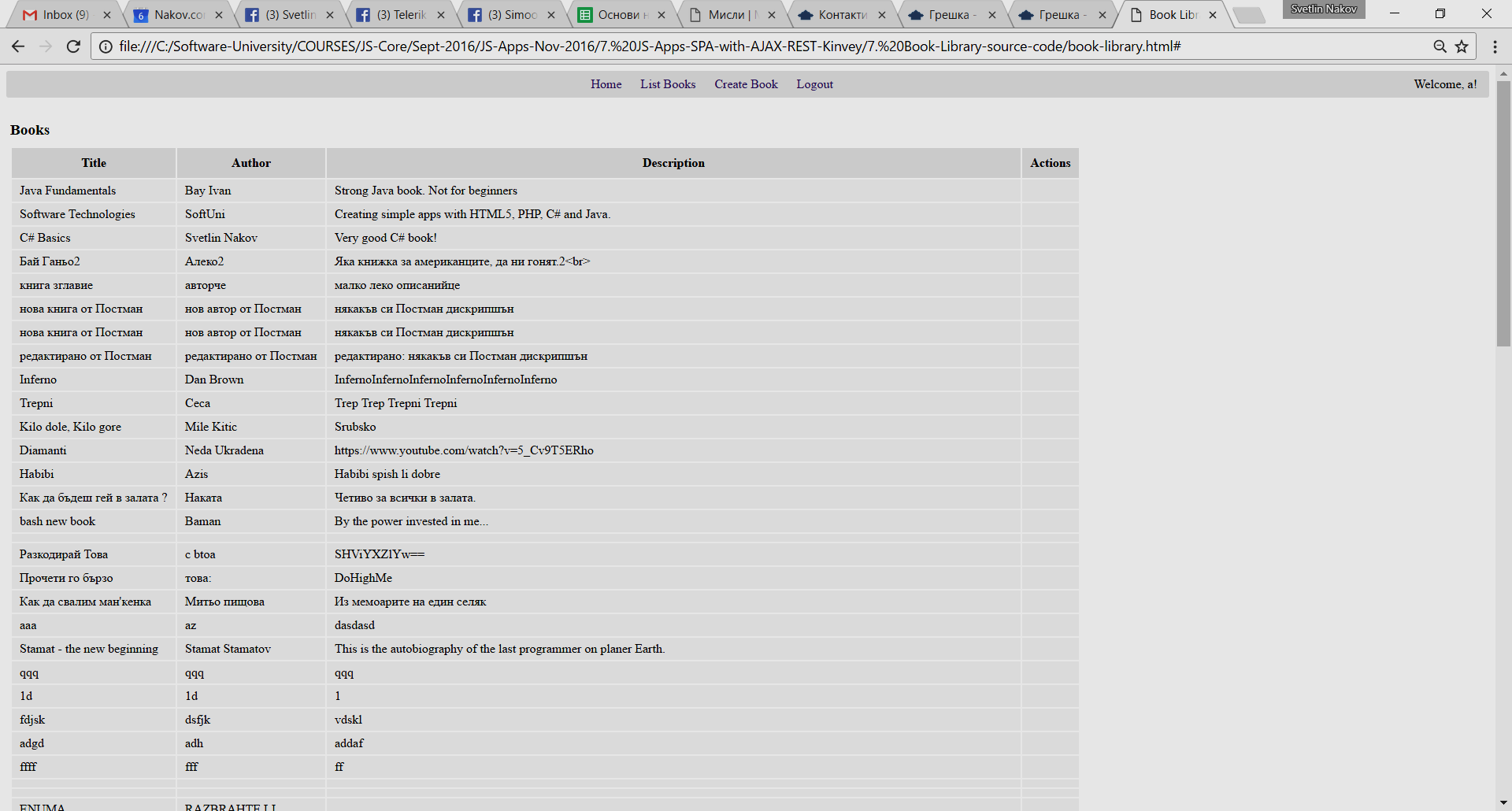
When you put **screenshots** in the document, use **small images** with **large fonts**.

* Only show the **essential part** of the screen.
* Use **large fonts**, so the text to be readable.
* **Keep images small**. Put several images on a single row when possible.

**Good screenshots** (home page + login form + user registration form – 3 small screenshots on a single row):

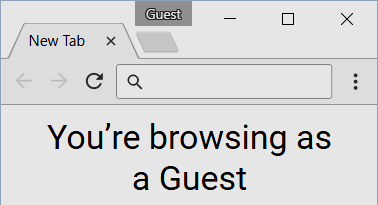
  

**Bad screenshots** example:



Hints for **better screenshots**:

* Use anonymous browser (**Guest session**), instead of logged in browser.

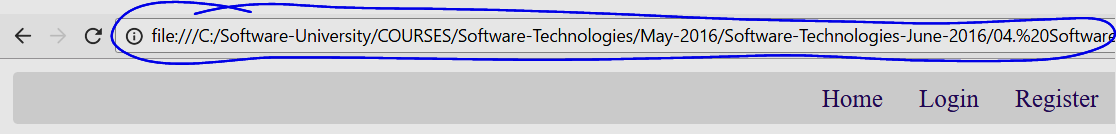


* You may take a **partial screenshot** (without the browser title) like this:

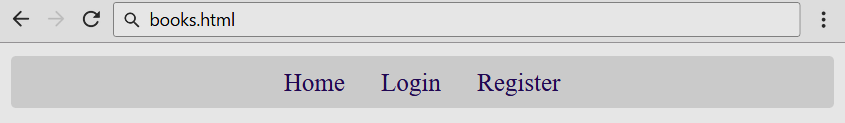


Such a screenshot illustrates very well the intended functionality and does not lose space.

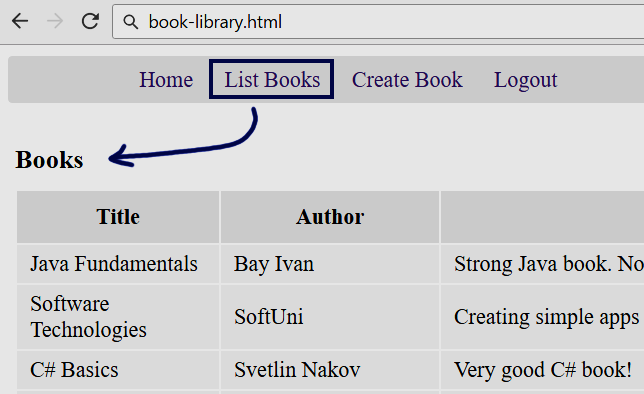
* Keep clean the **URL bar** in the browser. It should hold the file name without long paths. **Bad example**:



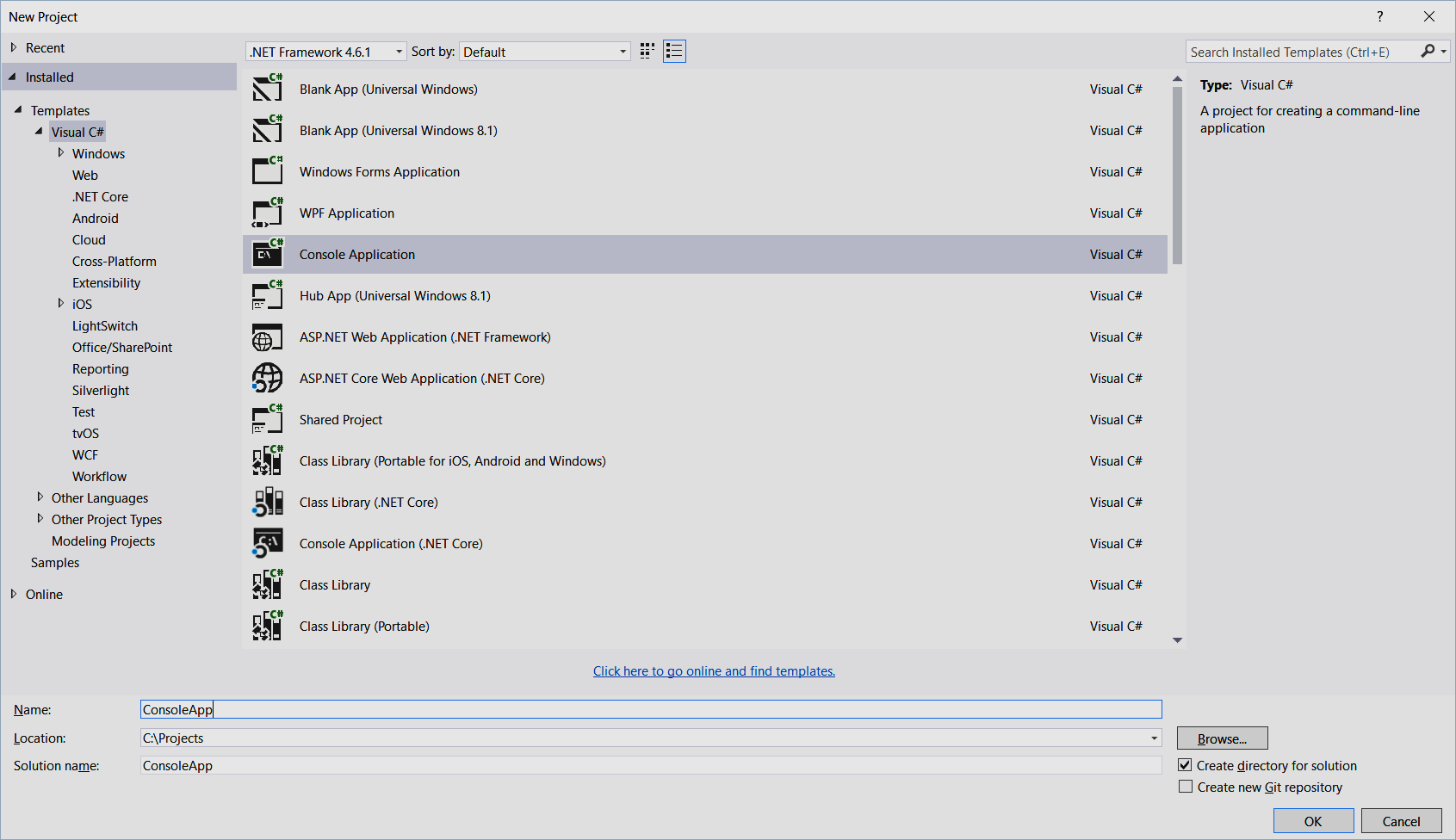
* The above may be fixed, by just **editing the URL bar** in the browser (yes, you may change it):



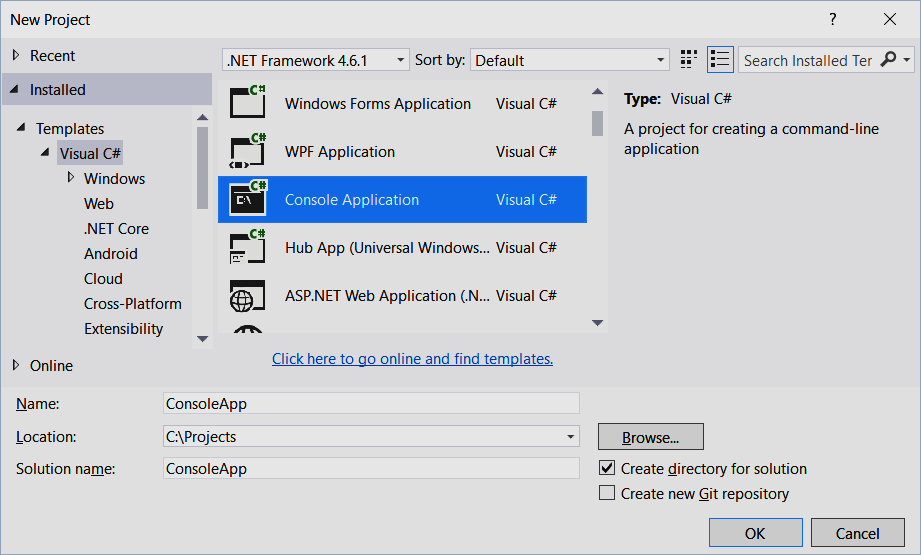
* Accent at certain screen areas using a **blue rectangle** like shown below. You may also use **blue arrows**. Use **dark** **blue color** (not red or green). You may use red for problems or errors (to show something wrong).



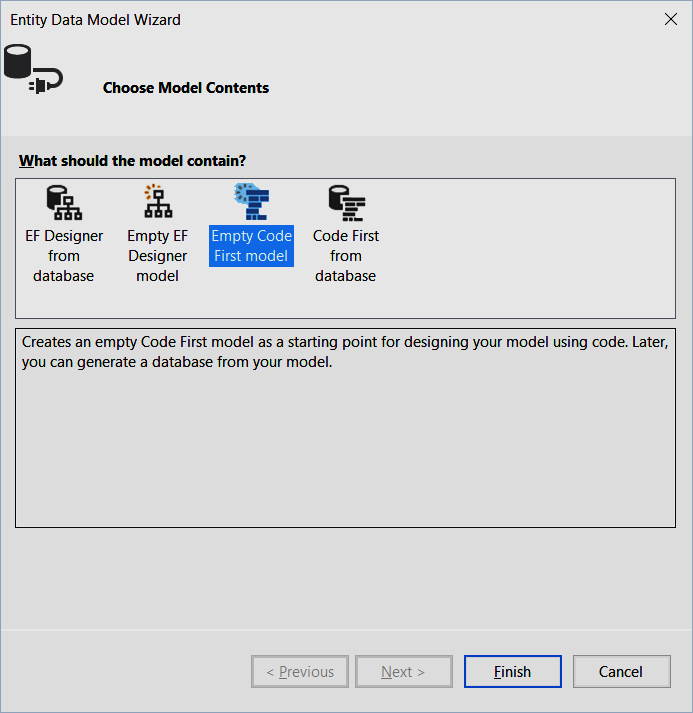
* Sometime you may **edit the screenshots** in some image editor (like [Paint.net](http://www.getpaint.net/)) to make them smaller.
* **Bad example** ofbig screenshot with **small unreadable text** (before editing), which loses a lot of space:



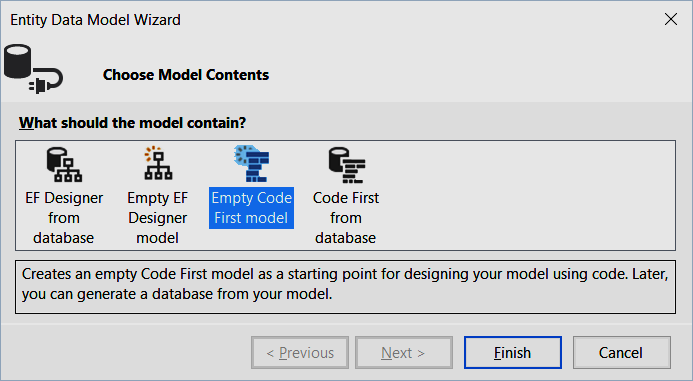
* **Good example** of smaller screenshot (after editing), showing the same screen using less space:



* Sometimes, even **the smallest possible window** taken as a screenshot is **too large**. In the following example, the window height is too much:



* The same screenshot can be **further edited in Paint.net** (or another image editor) to become smaller:



## Judge Tests

**Automate** the check-up for **as much as possible problems**, ideally for all of them.

* Use the **SoftUni Judge System** for automated testing.
* Use **input / output-based tests** for the console-style problems (text input + text output).
* Use SQL tests for DB problems, JS DOM tests for JS UI problems and HTML.
* Write custom **C# unit tests** / **Mocha unit tests** with mocking to check classes and other code that does not run through the console.

How to **design test cases**?

* Add the tests from the examples as **open tests** (input + output is visible after each submit).
* Add all other test cases as **closed tests** (hidden output, not visible for the students). These cases should cover the nominal (typical) cases and all special cases (or the most interesting of them in case of too much special cases).

## Typical Mistakes

Let’s review some **typical mistakes** made by the exam authors.

### Bad Problem Title

|  |
| --- |
|  |
| * Compare what? * Titles should be **short**, but **descriptive**. * A correct title might be the following: |
|  |
| * The above title if **over-descriptive**. It can be made shorted and still descriptive. * A correct title might be the following: |

### Bad Problem Title Formatting

|  |
| --- |
|  |
| * Keep **uppercase each word** in the document titles. * The correct formatting is the following: |

### Too Much Text in the Problem Description

|  |
| --- |
|  |
| * The **text is too long** and boring. * Avoid giving **unneeded stories**. * Explain the problem **by examples**, **figures** and **screenshots**, not by long text. * **Structure** the text better, e.g. with bullets. |

### Bad Text Formatting

|  |
| --- |
|  |
| * Keywords / key phrases should be formatted in **bold** to improve readability. * Variable names should be in “Code” style. * The **spell checker** should be configured to ignore the code elements (like variable names). * This is how the reformatted text looks like (**good example**): |

### Bad Table Formatting

|  |
| --- |
|  |
| * Incorrectly configured **spell checker**. * Incorrectly formatted **table**: incorrect margins and coloring. * Table **left border position** incorrect (the table should be positioned slightly on the left). * This is the **correctly formatted table** (good example): |

### Inappropriate Fonts / Colors / Styles

|  |
| --- |
|  |
| * Use **consistent formatting**. At SoftUni we use the same formatting guidelines and styles in all documents in the technical trainings: homework assignment, exams, tutorials, teamwork project descriptions, etc. * The fonts, colors and styles are already defined, just follow them. |

### Broken Page Footer

|  |
| --- |
|  |
| * **Page footer** (and also the document styles) might **get broken** when the document is edited by non-Microsoft editors like OpenOffice, Google Docs, Apple iWork or another tool. * Use the **original document template** for SoftUni and the original **Microsoft Word 2016** / 2012 / 2010. * Keep the documents in **DOCX format**, not DOC or ODT. |

### Missing Sample Input / Output for Console Programs

|  |
| --- |
|  |
| * This problem description is pointless without **sample input + output**. * It says nothing about many **special cases**: less than 3 numbers, 0 numbers, negative numbers, equal numbers, non-integers, etc. * The best way to fix this: give **sample inputs + outputs** which cover the nominal case and all special cases. |

### Missing Screenshots for UI Problems

|  |
| --- |
|  |
| Insert screenshots to give a visual idea what the student should build: |

### Bad Sample Input / Output

|  |
| --- |
|  |
| * Always give **several examples**. The sample input + output is only one, not enough for this problem. * It is not clear which number is **n** and which is **k** (the first or the second). * No example explains what happens if **n** < **k**. |
| An **improved set of examples** answer the above questions and explain what happens in all **special cases**. The below examples + comments explain the problem very well without need of detailed text explanations: |