# Mandatory exercise 3

This is a mandatory exercise. Make sure you deliver it in Canvas before the deadline. The exercise must also be approved at the lab.

Be aware that some of the questions deals with topics that we have not yet talked about in the lectures. You may wait to answer these questions after the lectures.

You may ask teachers and students for tips and advice, but in the end, it is **you** who must do the exercise so that you fully understand your solution and code. **At the approval of the exercise, you may be asked questions to verify that you fully understand your solution/code. If it is obvious that you don't understand your solution/code, the exercise will not be approved.**

## Task 1 – separate into different files

a) Open *foodlist.html*. Separate the JavaScript-code and the CSS-code into two new files: *foodlist.js* and *foodlist.css* – and "include" them to the HTML-file so that the program works in the same way as before.

b) Explain the difference between "including" a JavaScript-file - like you did in a) - and importing a *module*.

## Task 2 – regular expressions

a) Create a regular expression that we can use to test for a valid serial number in the format of:

**XX-xxx-xxx.xxxx**

Where ***x*** are numbers and ***X*** are letters (A-Z and a-z), for example: **AB-123-123.1234**

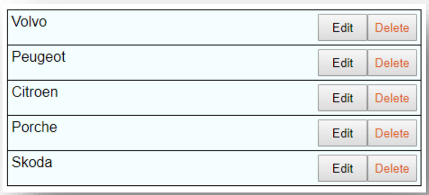
b) Create a regular expression that we can use to test for a valid Norwegian bank account number, i.e., it could be written in either of these formats:

| | |
|---|---|
| **xxxx.xx.xxxxx** | e.g., 1234.12.12345 |
| **xxxx xx xxxxx** | e.g., 1234 12 12345 |
| **xxxxxxxxxxx** | e.g., 12341212345 |

## Task 3 – Events, drag & drop

a) Create a list of <div>-elements based on the array shown below. Inside each div, there should be two buttons: *Edit* and *Delete*. The divs should be added to a container-div on the webpage:

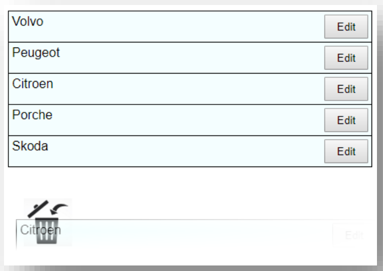
```
let cars = ["Volvo", "Peugeot", "Citroen", "Porche", "Skoda"];
```



Volvo Edit Delete
Peugeot Edit Delete
Citroen Edit Delete
Porche Edit Delete
Skoda Edit Delete

b) Write code so that when the user clicks on a *Delete*-button, the corresponding element in the array is removed, and the list is updated.

c) Change the program b) so that the user can drag an element in the list, and drop it on an image of a garbage-bin to delete it – instead of clicking on a Delete-button:



Note: nothing is supposed to happen when you click the Edit-buttons, i.e., the Edit-buttons are only for "show".

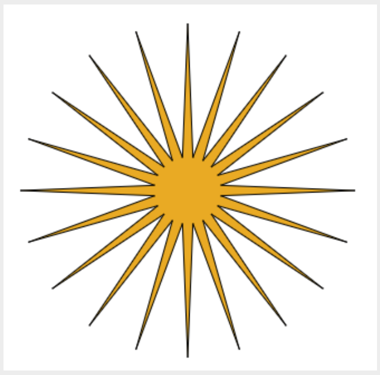
## Task 4 - OOP

Take a look at the code in the file *star.html* to see how you can draw a star in a Canvas-element.

Create a constructor-function or a class that draws a star in the center of a canvas element. The function or class should have the following interface:

**`Star(`*`cnv, n, r1, r2`*`) -`** Constructor where *cnv* is the canvas element, *n*, is the number of "spikes" on the star, *r1* is the inner radius, r2 is the outer radius, i.e. a bigger difference between r1 and r2 makes the spikes "sharper".

Additional methods in Star:
**`setNumSpikes(`*`n`*`)`** – sets the number of spikes
**`getNumSpikes()`** – gets the number of spikes
**`setOuterRadius(`*`r`*`)`** – sets the outer radius
**`getOuterRadius()`** – gets the outer radius
**`setInnerRadius(`*`r`*`)`** – sets the inner radius
**`getInnerRadius()`** – gets the inner radius
**`setLineColor(`*`color`*`)`** – sets the color of the outline
**`setFillColor(`*`color`*`)`** – sets the fill color



Example:
```
let myStar = new Star(myCnv, 20, 30, 150);
myStar.setFillColor("orange");
myStar.setLineColor("black");
```

## Task 5 – Promises

Explain these terms:

- Promise and promise states
- Async and await
- The *then* function

## Task 6 – MVC

Open *fetch_chocos_type.html*. Here we use *fetch* to contact a server and retrieve JSON-data with information about chocolates of a specific type. Change the program so that the code is separated into a *model*, *view,* and a *controller* (*MVC*):

a) **The view**: Create a *factory*-function named *createChocoListView* in a separate JS-file (name it *chocoview.js*) that returns a div-element containing the list. The div-element (object) should have a function:

- **showData(*arr*)** – show/refreshes the list with the data from the array *arr*.

In addition, you should create CSS code for the view in a CSS-file named *chocolistview.css.*

You can test the view by creating code in the HTML-file but remember to "link" the js-file in a separate <script>-tag. Then you can write code in another <script>-tag below. Here we create a new chocoListView-object, fills it with the data from the array and adds it to the webpage:

```
let chocoListView = createChocoListView();
chocoListView.showData(data); //data is the array with chocolates
document.body.appendChild(chocoListView);
```

b) **The model**: Create a *constructor*-function named *ChocoModel* in a separate js-file (name it *chocomodel.js*). *ChocoModel* should have the following interface:

- **ChocoModel()** – constructor.
- **getChocoData(*type*)** – returns a promise.

  Because downloading the data from the server is an asynchronous task, *getChocoData(type)* should return a promise. If the promise is resolved, it will contain the data.

  *type* is either *fyltsjokolade*, *rensjokolade* or *fudge*. The type must be added as an URL-variable in the server-URL.

c) **The controller**: In this example, you can write the code for the controller in the HTML-file (change the name to *controller.html*). You only need a few lines of code: create a new model-object, get the data, create a new view-object, and show the data. You should also output a message ("Sorry, no valid data") if the promise from *getData()* is rejected.

d) Sometimes we want the view to register user interaction, for example when the user clicks on one of the chocolates in the list. If we want to follow the MVC pattern, the view should not deal with this interaction. In addition, the view should (ideally) not refer to any objects outside itself. To solve this, we can let the view "forward" any user interaction by dispatching events. The controller can then register and handle these events.

Task: Continue with the code for the view (in *chocolistview.js*) and add an event handler for when the user clicks on one of the chocolates. Write code Inside the event handler for dispatching an event named *chocoselect*. Add a property to the event-object named *chocodata* that contains all the data for the chocolate that was clicked. Below is an example on how to register (use) the event in the controller:

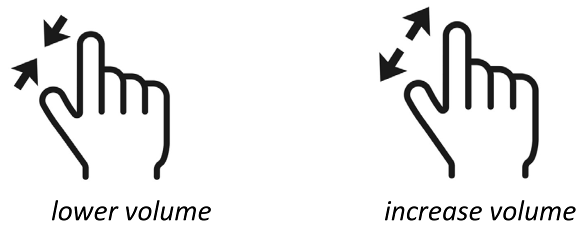
```
chocoListView.addEventListener("chocoselect", function(evt) {
      console.log(evt.chocodata);
});
```

e) (Non-mandatory) Create another view with another layout, for example show the chocolates in a grid with only the (square) images and the name and the price as text on top of the images. You can create another JS-file with another factory-function and name it *createChocoGridView().* You must also create another CSS-file for this view.
Also, create buttons at the top of the webpage so that you can switch between the views.

f) (Non-mandatory) Change the code so that you are using *modules*, i.e., the view, the model and the controller should be in separate module-files. Be aware that the files must be on a (local) server to make this work. In Visual Studio Code you can add an Extension named *Live Server* to test the app on a server.

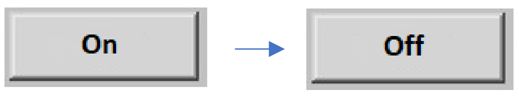
## Task 7 – UI and Interaction design

a) Peter is creating a music player app and he consider controlling the volume by the following touch gestures:

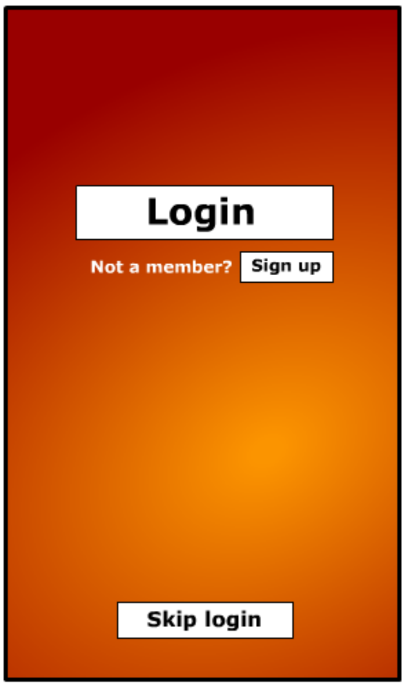


Should Peter implement the gestures? Why or why not?

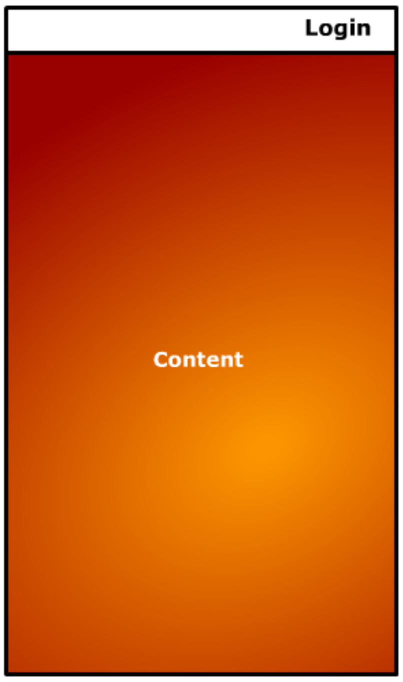
b) You are creating a user interface component which should turn something on and off. Is it OK to use a (single) button where the text changes to indicate if the button is turned on or not? Why or why not?



c) Which of the following layouts for an opening screen would you choose? Why?

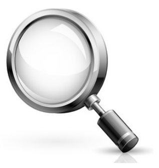


A

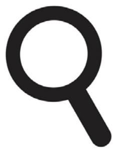


B

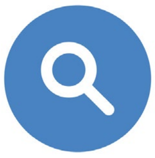
You are implementing a search-button in your app. What kind of icon would you use? Why?



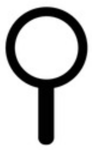
A



B



C



D

d) What is negative space? How can we use it to make a better user interface?
e) What is *onboarding*? Should we always use it – or try to avoid it?

## Task 8 – Single-page app - ViewSwitcher (not mandatory)

Create a constructor-function or a class named *ViewSwitcher* for switching between views ("pages") in an app. The views should be defined as <template> elements on the webpage and each view should have the same width and height as the screen. ViewSwitcher should have the following interface:

- ViewSwitcher(*container*) – constructor. *container* is a reference to an element (usually a <div>) on the webpage that will contain the views.
- addView(*id*) – creates a new view in the ViewSwitcher based on a <template>-element with the given id.
- showView(*id*) – shows the view with the given id inside the container. When you show a view, any pre-existing view will be hidden.
- removeView(*id*) – deletes a view from the ViewSwitcher.

Tip: here is an example on creating DOM-elements out of a template and add them to the webpage:

HTML:
```
<template id="page1">
    <div>
       <h2>My nice heading</h2>
       <p></p>
    </div>
</template>
```

JavaScript:
```
let tmpl = document.getElementById('page1');
tmpl.content.querySelector("p").innerHTML = "My nice paragraph";
let page = tmpl.content.cloneNode(true);
document.body.appendChild(page);
```

**Animation**
If you want, you can try to add animation when you show a new view (e.g., by scaling or moving). Here is an example on a CSS-animation and on how to play it from JavaScript:

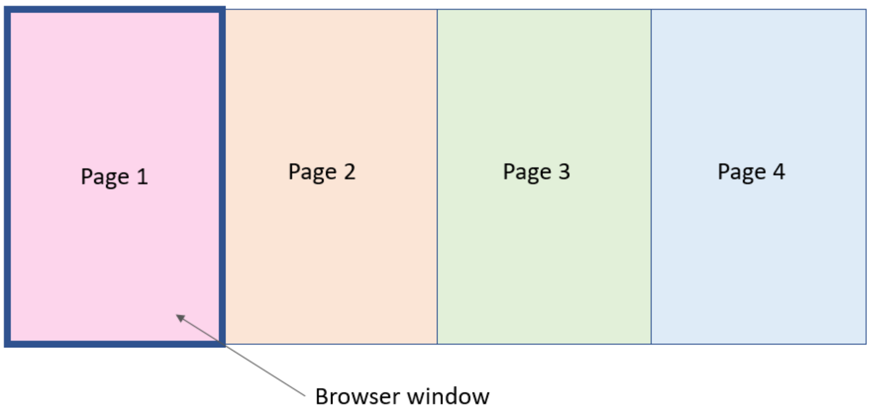
CSS:
```
@keyframes pageani {
    from {transform: scale(0.1, 0.1);}
    to {transform: scale(1, 1);}
}

.aniplay {
    animation-name: pageani;
    animation-duration: 0.5s;
    transform-origin: 0% 0%;
    transform: scale(1, 1);
}
```

Play animation from JavaScript
```
page.classList.remove("aniplay");
void page.offsetWidth; //hack for restart (force DOM reflow)
page.classList.add("aniplay");
```

## Task 9 – Single-page app - ViewSwiper (not mandatory)
Create a constructor-function or a class named *ViewSwiper* for swiping horizontally between views ("pages") in an app. The views should be defined as <template> elements on the webpage and it should have the same interface as *ViewSwitcher*. In addition, you could add functionality to change the order of the views.

Tip: Each view should have the same width and height as the screen. Add them to a container-element so every view is placed next to each other (horizontally):



The easiest way to implement swiping is to use CSS to scroll-snapping between the views. Be aware that in a "non-mobile" browser you must use the horizontal scrollbar – instead of "swiping" with the mouse:

- In the CSS for the container element:
  ```
  overflow-x: scroll;
  scroll-snap-type: x mandatory;
  ```
- In the CSS for the view-elements:
  ```
  scroll-snap-align: start;
  ```

You could also add a navigation bar that shows which page is active: