|  |  |
| --- | --- |
| **Exercise** | JS.1 |
| **App** | **Part1Exercise** |
| **Purpose** | Try to keep track of variable environment and execution contexts. |
| **Description** | The **script.js** file contains a bit of code to perform some slightly convoluted calculations. |
| **Steps** | Before actually running the code, see if you can figure out what the expected output should be, i.e. what the values of **res1**, **res** 2 and **res3** are when they are printed out to the console. |

|  |  |
| --- | --- |
| **Exercise** | JS.2 |
| **App** | **Part2Exercise** |
| **Purpose** | Try to work with object building |
| **Description** | Imagine that we are taking the first small steps towards building a sort of role-play game. In the game, there will be “participants”. A parti­cipant has a name, and a number of “hit points”. There will be two types of participants:   * **Hero**: also has a “role” (like e.g. Wizard or Hunter), a “level” (from 1 and up), and a number of gold coins (a numeric value) * **Beast**: also has a “value”, which is a numeric value |
| **Steps** | Write JS code to enable creation of **Hero** and **Beast** objects, given the above spe­ci­fications. You can use any style you prefer, also including elements from ES6. Remember to test your code a bit as well ☺. |

|  |  |
| --- | --- |
| **Exercise** | JS.3 |
| **App** | **Part3Exercise** |
| **Purpose** | Work with array processing methods like **map**, **filter** and **reduce** |
| **Description** | The **script.js** file contains a number of small exercises, which should be solved using **map**, **filter** and **reduce** |
| **Steps** | Open **script.js**, and try to solve exercise #1 to #7  Note that when you run the JS code, you should put in a breakpoint just after the part of the code you are testing. Run the code in debug mode; it will then pause at the breakpoint, and you can inspect the objects printed by **trueLog** in the **Debug Console** window. |

|  |  |
| --- | --- |
| **Exercise** | JS.4 |
| **App** | **Part4Exercise** |
| **Purpose** | Work with closures and function factories |
| **Description** | The **script.js** file contains a number of small exercises, which involve closures and function factories. |
| **Steps** | Open **script.js**, and try to solve exercise #1 to #4 |

|  |  |
| --- | --- |
| **Exercise** | JS.5 |
| **App** | **Part5Exercise** |
| **Purpose** | Work with the **fetch** function, and with the asynchronous program­ming paradigm in general. |
| **Description** | The project contains a small app for retrieving the image of GitHub users. More specifically, the user can type in a GitHub account name, and the image used on that GitHub account is then displayed. This is achieved in three steps:   1. Retrieve the account name typed into the input control. 2. Retrieve data from the GitHub account. 3. Use the retrieved data to set the image source for the image control.   However, step 2 and 3 has not been implemented yet. |
| **Steps** | 1. Open **script.js**, and inspect the code. It is as such quite simple, but it only implements step 1. 2. Implement step 2 using the **fetch** function (see e.g. the Mozilla ducumentation on **fetch**: <https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using_Fetch>). You can choose to use the style with **then**-functions, or alternatively the style with **async**/**await** (or you can try out both approaches). 3. Implement step 3 using the retrieved data and the provided function **setImageFromUserData**. 4. Try out your code with a couple of GitHub user accounts, e.g. “perl-easj”, “laursen”, “billgates” and “octocat”. |

|  |  |
| --- | --- |
| **Exercise** | JS.RolePlay |
| **App** | **RolePlay** |
| **Purpose** | Try to implement a very small role-play game in JavaScript |
| **Description** | In JS.2, we took some small steps towards building a small role-play game in JavaScript. In this exercise, we aim at implementing a very simple logic for two game participants fighting against each other.  More specifically, try to extend the **Hero** class with the below pro­perties and functions:   * A “health points” property * A method for “dealing damage”. This method should return a numeric value. * A method for “receiving damage”, which will cause the health points of the Hero to be reduced. * A method for deciding if the Hero is dead or not, dead being defined as having zero or less health points   A match between two Heroes should follow this simple logic:  While both Heroes are alive:   1. First Hero deals damage; the dealt damage is received by the second Hero. 2. Second Hero (if still alive) deals damage; the dealt damage is received by the first Hero. |
| **Steps** | 1. Extend the Hero class – in whatever way your prefer to define it – to include the functionalities described above. 2. Define and use a function that can execute a match between two Heroes. You can test this by doing simple **console.log** calls in the JS code. 3. Once the game logic works, try to create a simple GUI for the game, by adding HTML to the **index.html** file. The GUI should contain four elements:    * A button for letting the first Hero attack the second Hero.    * A button for letting the second Hero attack the first Hero.    * A button for resetting the game to an initial state.    * A status text describing the current state of the game (how many health points do the Heroes have left, etc..) |

|  |  |
| --- | --- |
| **Exercise** | JS.RolePlayClientServer |
| **App** | **RolePlayClientServer** (and **RolePlaySolved**) |
| **Purpose** | See and work with an example of a client/server setup using **Node.js**. |
| **Description** | The **RolePlaySolved** folder contains a very simple role-play game, where a **Hero** and a **Beast** can fight to the death. In this exercise, we take a look at a client/server implementation of the game. That is, the game runs in a **Node.js**-based server application, and can be accessed by a client through the browser. |
| **Steps** | 1. Take a thorough look at the code in **RolePlaySolved** (primarily the code in **script.js**), until you feel you understand the structure of the code, and the role of each class. 2. Now open the **RolePlayClientServer** folder, and take a look at the code in **app.js**. This code is the “server part” of the game. You should primarily focus on understanding the code beneath the comments **// Game setup** and **// Server setup**. 3. Also take a look at the code in **script.js**. This code is the “client part” of the game. You should primarily focus on understanding the code in the classes **GameServer­Connection** and **GameClientLauncher**. 4. Start the server, by opening a terminal window, navigate to the root of the **RolePlayClientServer** folder (if needed), and enter **NODE app.js**. This starts the server. **NB:** Keep the terminal window open! When you wish to shut down the server, go back into the terminal window and type **Ctrl + C**. 5. Now open **index.html** (e.g. with **Open with Live Server**) and try to play the game. Try to open the browser **DevTools** window, and see the network traffic caused by playing the game. 6. Update implementation to use Axios. 7. The game is slightly weird currently, since the user has complete control over the order of attacks. A variant of the game could be to just have a single “Attack!” button, which would cause the next attack to occur. This should proceed such that the participants take turns attacking each other. How will this change the game logic, the game state, and the implementa­tion of the game? See if you can implement such a variant of the game. |