# Exercises: Modules, Babel and CommonJS

Problems for exercises and homework for the [“JavaScript Advanced” course @ SoftUni](https://softuni.bg/courses/javascript-advanced). Submit your solutions in the SoftUni judge system at <https://judge.softuni.bg/Contests/343/>.

## People Modules

Revisit the problem **People** from **Class Inheritance**. Refactor the code so each class is a separate module in its own file. Create an app.js file that loads all class definitions and attaches them to the result property.

Define several JS classes, that represent a company’s employee records. Every employee has a **name** and **age**, a **salary** and a list of **tasks**, while every position has specific properties not present in the others. Place all common functionality in a **parent** **abstract** class. Follow the diagram bellow:



Every position has different tasks. In addition to all common properties, the manager position has a **dividend** he can collect along with his salary.

All employees have a **work** function that when called cycles trough the list responsibilities for that position and prints the current one. When all tasks have been printed, the list starts over from the beginning. Employees can also **collect salary**, which outputs the amount, plus any **bonuses**.

Your program needs to expose a module, containing the three classes Junior, Senior and Manager. The properties name and age are set trough the constructor, while the salary and a manager’s dividend are initially set to zero and can be changed later. The list of tasks is filled by each position. The resulting objects also expose the functions work() and collectSalary(). When work() is called, one of the following lines is printed on the console, depending on the current task in the list:

{employee name} is working on a simple task.

{employee name} is working on a complicated task.

{employee name} is taking time off work.

{employee name} is supervising junior workers.

{employee name} scheduled a meeting.

{employee name} is preparing a quarterly report.

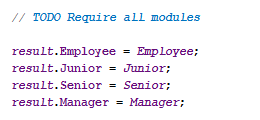
And when collectSalary() is called, print the following:

{employee name} received {salary + bonuses} this month.

### Input / Output

Any input will be passed as valid arguments, where applicable. Print any output that is required to the console as a string.

Submit your code as a zip archive, containing all four modules and an **app.js** file that attaches them to the **result** property. Name the four classes and any properties exactly as described above. This is how your code might look like:



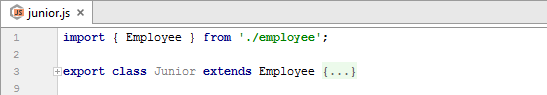
### Hints

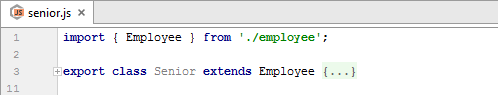
For help with defining the classes, check out the problem description from the [Class Inheritance Exercise](https://softuni.bg/downloads/svn/js-core/Sept-2016/JS-Advanced-Oct-2016/14.%20JS-Advanced-Inheritance-Unit-Testing-Exercises/13.%20JS-Advanced-Inheritance-Unit-Testing-Exercises.docx). If you have a working solution, you can copy it and use it as a starting point. We will demonstrate a solution using **ES6 syntax** but you can write it using **CommonJS** if you’re more comfortable (both methods will work in the Judge).

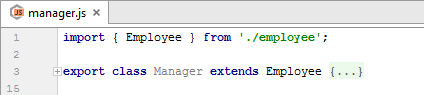
The first thing we need to do is place the code for each class in a separate file (module). To expose the base class Employee, so that the other three modules can import and extend it, we add the keyword export before the class definition.



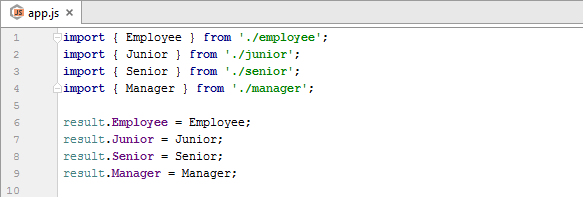
Now the other modules can import the class definition. Since all three other classes extend Employee, we add a reference to it in all three modules. To do this, on the first line of each file, we add an import statement with a relative path to the Employee module (file). We also need to export the class definitions themselves, so we add the export keyword before each.







Now that we have exposed all classes, we create an **app.js** file that loads them all and attaches them to the result property.



Note that if you want to test your solution locally, your code must first be **transpiled** using Babel. Review the lecture presentation for setup instructions.

## Branches

You have been tasked to create JavaScript classes for a marketing system which operates over several companies. The classes you need to create must be, each in a different file. To connect the functionality of each of the classes, you will need to export them as modules.

You need to create a class Branch, which is initialized with parameters – id (Number), branchName (String) and companyName (String). Those arguments’ values should be stored in **private members** for the class.

The Branch class should also have a list of employees in which it **stores all of its employees**. That list should have a **getter** called employees, which should return the list directly.

The Branch class should also have a function hire(employee) which **adds a given employee** to the list of employees.

You also need to implement a toString() function, which represents the class as a string. The function should return data in the following format:

“@ {companyName}, {branchName}, {branchId}”  
“Employed:”  
“\*\* {employee1}”  
“\*\* {employee2}”  
. . .

In case there are no employees, just print “None…” below “Employed:”.

You also need to create a class Employee. The Employee class is initialized with a name (String), age (Number) and a position (String). Those arguments’ values should be stored in **public** **members** for the class.

The Employee class should also have a toString() function, which returns data about the class in the following format:

“{name}, {age} ({position})”

The toString() function of the employee should be called when presenting a **Branch’s employees**.

### Example

|  |
| --- |
| script.js |
| **let** employee1 = **new** Employee(**"Peter Ivanov"**, 25, **"Regular Worker"**); **let** employee2 = **new** Employee(**"Ivan Peterov"**, 24, **"Regular Worker"**); **let** employee3 = **new** Employee(**"John Hissie"**, 21, **"CEO"**);  **console**.log(employee3.toString()); *// John Hissie, 21 (CEO)* **let** branch1 = **new** Branch(1, **"Four Streets Branch"**, **"Default INC."**); branch1.hire(employee1); branch1.hire(employee2); branch1.hire(employee3); **console**.log(branch1.toString()); *// Employed: // \*\* Peter Ivanov, 25 (Regular Worker) // \*\* Ivan Peterov, 24 (Regular Worker) // \*\* John Hissie, 21 (CEO)* |

## Turtles

You have been asked to create the class hierarchy for a game of turtles.

You need to create a base class for the turtles, called Turtle, obviously…

The Turtle class should be initialized with a name (String), age (Number), and a gender (String). Each of those arguments’ values should be stored as a **private** **member**. Each of those private members should have **getters**, named **exactly** as the **arguments**.

As a base class the Turtle should be an **abstract** class and should not be instantiated directly. Make sure you **throw** an **error** if someone tries to create an instance of the Turtle class.

The Turtle class should have a function grow(age) which **increases the value** of the **age** property with the given parameter.

Implement a toString() function for the base class, which returns the following result:

“Turtle: {turtleName}”  
“Aged – {turtleAge}; Gender – {turtleGender}”

There are several other classes you need to implement.

Implement a class WaterTurtle, which inherits the base class.

The WaterTurtle class, should be initialized with an additional parameter – waterPool (String), which is the natural place at which the turtle currently is. The argument’s value should be stored in a **private member** for the class. There should be a getter called currentWaterPool which returns its value.

The WaterTurtle class should also hold a travel(waterPool) function, which changes the **current water pool’s** **value** with the **given one**. Traveling takes time, so the turtle should **grow 5 years** upon traveling to another water pool.

The WaterTurtle should extend the base toString() functionality, adding to it the following line:

“Currently inhabiting {currentWaterPool}”

The next turtle in line is the GalapagusTurtle. Implement a class for that turtle, which inherits the base class and is initialized with the **same arguments** as the base class.

The GalapagusTurtle should have a list of things it has eaten **THIS YEAR**, which is **initially** **empty**. There should be a **getter** for that list, called thingsEaten.

It should also have an eat(food) function which **adds** a given food (String) to that list. Upon growing, the year passes so that list should become **empty** for the new year. Override the base grow() function to achieve that.

The GalapagusTurtle extends the base toString() functionality, adding to it the following line:

“Things, eaten this year: {food1}, {food2}, {food3}…”

The next turtle is the EvkodianTurtle. Implement a class for that turtle, which inherits the base class, and is initialized with an additional parameter called evkodiumValue.

The EvkodianTurtle should hold a property evkodium, which only has a **getter**. The getter should return an object holding two things

value: {evkodiumValue} – the value given to the constructor.

density: {age \* 3} if the turtle is a “male” and {age \* 2} if it is a “female”.

The EvkodianTurtle should extend the base toString() functionality adding to it the following line:

“Evkodium: {evkodiumValue \* density}”

There is one more turtle you need to implement. That is the NinjaTurtle. As all other, make it inherit the base class.

The NinjaTurtle class should be initialized with 2 extra parameters – maskColor (String), and weapon (String).

Upon **growing**, the ninja turtle **increases its level**, with the **age** it has grown, starting for **0 initially**.

That affects the toString() function of the NinjaTurtle. The function should extend the base functionality and add the following line:

“{first three letters of name} wears a {maskColor} mask, and is an apprentice with the {weapon}.” – if the **level** of the turtle is below 25.

“{first three letters of name} wears a {maskColor} mask, and is smokin strong with the {weapon}.” – if the **level** of the turtle is **more** or **equal** to **25** and **less** than **100**.

“{first three letters of name} wears a {maskColor} mask, and is BEYOND GODLIKE with the {weapon}.” – if the **level** of the turtle is **more** or **equal** to **100**.

### Example

|  |
| --- |
| script.js |
| **let** testWaterTurtle = **new** WaterTurtle(**"Michelangelo"**, 18, **"male"**, **"Sewer"**); **let** testGalapagosTurtle = **new** GalapagosTurtle(**"Raphael"**, 18, **"male"**); **let** testEvkodianTurtle = **new** EvkodianTurtle(**"Donatello"**, 18, **"male"**, 100); **let** testNinjaTurtle = **new** NinjaTurtle(**"Leonardo"**, 18, **"male"**, **"Blue"**, **"Yamato"**);  **console**.log(testWaterTurtle.toString()); *// Turtle: Michelangelo // Aged - 18; Gender - male // Currently inhabiting Sewer* **console**.log(testGalapagosTurtle.toString()); *// Turtle: Raphael // Aged - 18; Gender - male // Things, eaten this year:* **console**.log(testEvkodianTurtle.toString()); *// Turtle: Donatello // Aged - 18; Gender - male // Evkodium: 5400* **console**.log(testNinjaTurtle.toString()); *// Turtle: Leonardo // Aged - 18; Gender - male // Leo wears a Blue mask, and is an apprentice with the Yamato.* |

## People and Posts

Write two classes Person and Post, the classes should support the following functionality:

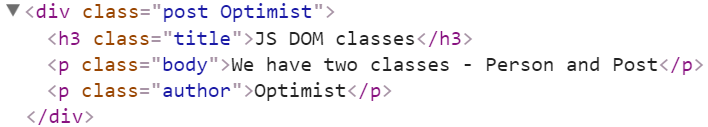
* Person
  + name - a string property containing the person's name
  + age - a number property containing the person's age
  + addToSelector(selector) - a function that attaches a **COPY** of the DOM representation of the person to every element matching the passed in selector. The DOM representation should contain the following:
    - A div with class="person {person.name}", the div should contain the following:
      * A paragraph with class="name" and text equal to the name property of the object.
      * A paragraph with class="age" and text equal to the age property of the object.
      * A div for the posts of the person with class="posts {person.name}".
* Post
  + title - a string property containing the post's title
  + body - a string property containing the post's body
  + author - a string property containing the post's author
  + addToSelector(selector) - a function that attaches a **COPY** of the DOM representation of the post to every element matching the passed in selector. The DOM representation should contain the following:
    - A div with class="post {post.author}", the div should contain the following:
      * A heading level 3 (<h3>) with class="title" and text equal to the post's title.
      * A paragraph with class="body" and text equal to the post's body.
      * A paragraph with class="author" and text equal to the post's author.

### Screenshots

Example of a person object's DOM representation:

### 

Example of a post object's DOM representation:



### Example

Here's an example template of how your code should look:

|  |
| --- |
| template.js |
| *//Require/Import the Person and Post classes* ***result***.**Person** = *Person*;  ***result***.**Post** = *Post*; |

## Checkboxes and Numberboxes

Write two classes Checkbox and Numberbox that represent single DOM elements - a Checkbox object represents an input type="checkbox" and a Numberbox object represents an input type="number". The classes should support the following functionality:

* Checkbox
  + constructor(label, selector)
  + label - a string property with a getter, representing the label for the input.
  + elements - a getter for the objects matched by the selector
  + value - a Boolean property with a setter and getter, the setter also has validation that the passed in parameter is of type Boolean. In case of incorrect input throws and Error. A new object of type Checkbox should start with a default value that of the DOM checkbox it's connected to.
* Numberbox
  + constructor(label, selector, minValue, maxValue)
  + label - a string property with a getter, representing the label for the input.
  + elements - a getter for the objects matched by the selector
  + value - a number property with a setter and getter, the setter also has validation that the passed in parameter is a valid integer between the min and maxValue inclusive. In case of incorrect input throws and Error. A new object of type Numberbox should start with a default value of minValue.

An object of type Checkbox's value and the DOM checkbox element representing it have their **values linked** - whenever the object's value changes the DOM element should instantly mirror it and vice versa. A Numberbox is linked to its input type="number" in the same way.

### Constraints

* Selectors will always point to single input elements of their correct types.

### Example

To help you test your code, you're provided with an **HTML** template:

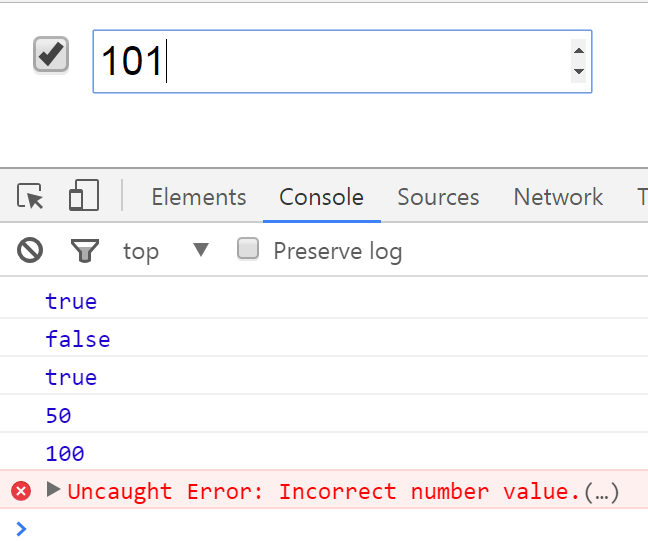
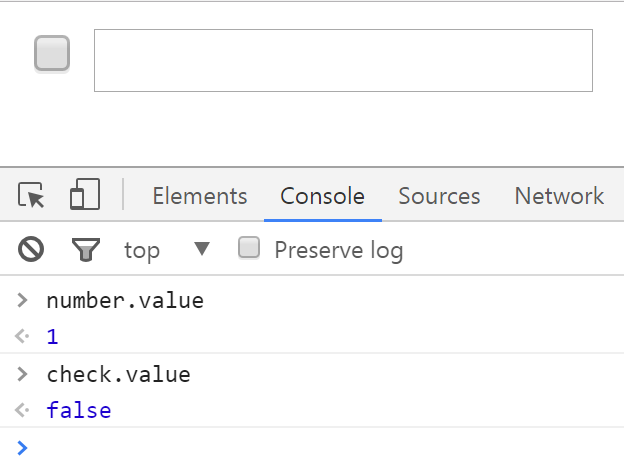
|  |
| --- |
| test.html |
| <!DOCTYPE **html**> <**html lang="en"**> <**head**>  <**meta charset="UTF-8"**>  <**title**>Checkbox</**title**>  <**script src="https://code.jquery.com/jquery-3.1.1.min.js"**></**script**> </**head**> <**body**>  <**div id="wrapper"**>  <**input id="married" type="checkbox"**/>  <**input id="age" type="number"**/>  </**div**>  <**script src="app.js"**></**script**> </**body**> </**html**> |

You can have temporarily put your classes in the app.js in order to test:

|  |
| --- |
| app.js |
| **class *Checkbox*** {...}  **class *Numberbox*** {...}  **let *check*** = **new *Checkbox***(**"Is Married:"**,**"#married"**); **let *number*** = **new *Numberbox***(**"Age:"**,**"#age"**,1,100); **let *checkbox*** = $(**'#married'**); **let *numberbox*** = $(**'#age'**); ***checkbox***.on(**'change'**,()=>**console**.log(***check***.value)); ***numberbox***.on(**'change'**,()=>**console**.log(***number***.value)); |

### Screenshots

Here are some examples in the browser:

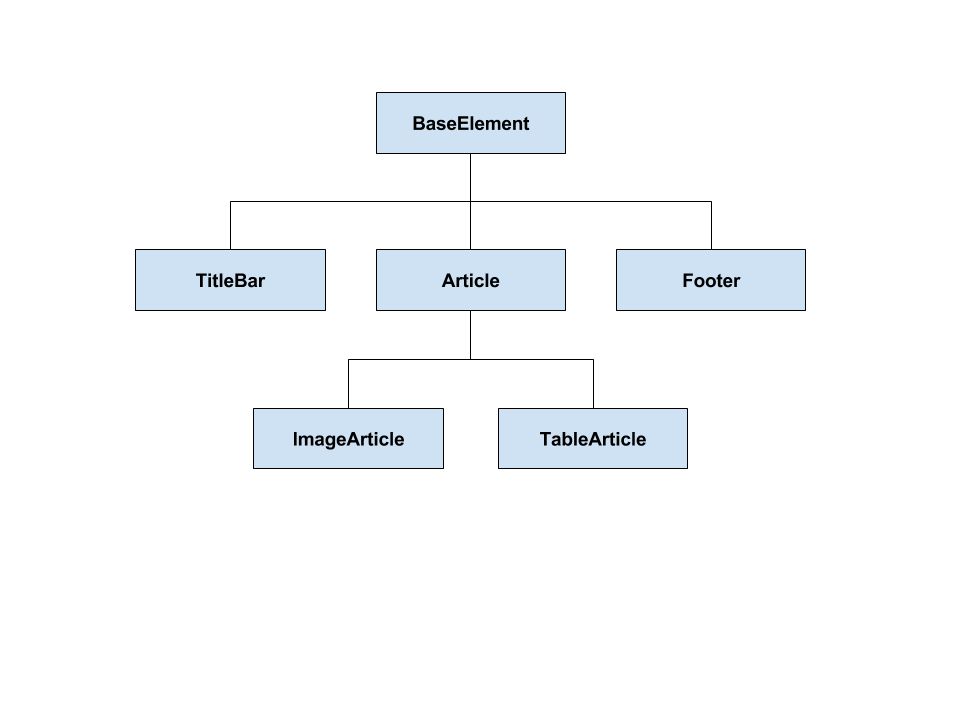


You need to submit a **zip** archivewhich attaches the two classes Checkbox and Numberbox to the result variable.

|  |
| --- |
| temlate.js |
| **//Require/Import classes**  **result.*Checkbox*** = Checkbox;  **result.*Numberbox*** = Numberbox; |

## Modular DOM

Write a framework of UI elements for use in a browser. Write an abstract Base Element class with common functionality and child classes that extend it for a Title Bar, an Article and a Footer. The Article is further extended by classes for Image Article and Table Article. Follow the diagram bellow for class hierarchy.



The BaseElement class should be **abstract**. It has an element property that stores a jQuery element which is set to null at instantiation. The method appendTo(selector) calls createElement() to initialize **element** and appends it to the passed in **selector**. createElement() sets the value of element to the value returned by getElementString().

|  |
| --- |
| base-element.js |
| BaseElement {  element, // jQuery element  appendTo(selector) // call *createElement()* and append *element* to selector  createElement() // initialize *element* to result of *getElementString()*  getElementString() // return HTML as string  } |

The TitleBar class extends BaseElement. It has a title that is set trough the constructor and a list of links that is initially an empty array. The method addLink(href, name) adds a link to the list. You should override getElementString() to return the HTML of the template provided bellow.

|  |
| --- |
| title-bar.js |
| TitleBar {  title, // documen title  links, // list of navigation links  addLink(href, name) // add link to navigation  getElementString() // override base method to return correct HTML  } |

Everything in curly braces should be replaced when you generate your HTML. Note there will be a varying number of menu links.

|  |
| --- |
| Title Bar HTML |
| <**header class="header"**>  <**div**><**span class="title"**>{Title Bar Problem}</**span**></**div**>  <**div**>  <**nav class="menu"**>  <**a class="menu-link" href="{href}"**>{Name}</**a**>|  <**a class="menu-link" href="{href}"**>{Name}</**a**>|  <**a class="menu-link" href="{href}"**>{Name}</**a**>  </**nav**>  </**div**> </**header**> |

The Footer class extends BaseElement. It has a message that is set trough the constructor. You should override getElementString() to return the HTML of the template provided bellow.

|  |
| --- |
| footer.js |
| Footer {  message, // copyright message  getElementString() // override base method to return correct HTML  } |

Everything in curly braces should be replaced when you generate your HTML.

|  |
| --- |
| Footer HTML |
| <**footer**>Copyright **&copy;** {Message}</**footer**> |

The Article class extends BaseElement. It has a title and content that are set trough the constructor and a timestamp that is automatically set to the **current time** when the object is instantiated. You should override getElementString() to return the HTML of the template provided bellow.

|  |
| --- |
| article.js |
| Article {  title, // article title  content, // article content  timestamp, // creation time, use *new Date()*  getElementString() // override base method to return correct HTML  } |

Everything in curly braces should be replaced when you generate your HTML.

|  |
| --- |
| Article HTML |
| <**div class="article"**>  <**div class="article-title"**>{Article title}</**div**>  <**p**>{Content … }</**p**> </**div**> |

The ImageArticle class extends Article. It has an image that is set trough the constructor in addition to other parameters, required for the base constructor. You should override getElementString() to return the HTML of the template provided bellow.

|  |
| --- |
| iamge-article.js |
| ImageArticle {  image, // image source  getElementString() // override base method to return correct HTML  } |

Everything in curly braces should be replaced when you generate your HTML.

|  |
| --- |
| Image Article HTML |
| <**div class="article"**>  <**div class="article-title"**>{Article title}</**div**>  <**div class="image"**><**img src="{src/lorem.png}"**></**div**>  <**p**>{Content … }</**p**> </**div**> |

The TableArticle class extends Article. It has a headings and data properties that are initially set to null in addition to other parameters, required for the base constructor. The loadData(headings, data) method sets the values of **headings** and **data** to the passed in parameters. You should override getElementString() to evaluate the **headings** and **data** and return the HTML of the template provided bellow. The **headings** will be an array of strings, each element is a heading. The **data** is an array of object with each object a row of the table and each property a column of the row. The headings will match the names of the properties, but will be capitalized and ay contain spaces – you need to make them lowercase and remove any white space to access the properties.

|  |
| --- |
| table-article.js |
| TableArticle {  headings, // table headings  data, // table data  loadData(headings, data), // set *headings* and *data*  getElementString() // override base method to return correct HTML  } |

Everything in curly braces should be replaced when you generate your HTML. Note there will be a varying number of table rows and columns.

|  |
| --- |
| Table Article HTML |
| <**div class="article"**>  <**div class="article-title"**>{Article title}</**div**>  <**p**>{Content … }</**p**>  <**div class="table"**>  <**table class="data"**>  <**tr**><**th**>{Entry Header 1}</**th**>{ … }</**tr**>  <**tr**><**td**>{Entry First Line 1}</**td**>{ … }</**tr**>  { … }  </**table**>  </**div**> </**div**> |

There will be a project skeleton provided with an example HTML, containing a finished page (with no scripts), a shell HTML where you can test and a sample app.js, containing commands that should generate the sample page. Note that you will have to include any scripts manually.