Exercises: Prototypes and Interitance

Problems for exercises and homework for the "JavaScript Advanced" course @ SoftUni. Submit your solutions in the SoftUni judge system at https://judge.softuni.bg/Contests/1677/Exercise-Prototypes-and-Inheritance

1. Array Extension

Extend the build-in **Array** object with additional functionality. Implement the following functionality:

- **last()** - returns the last element of the array
- **skip(n)** - returns a new array which includes all original elements, except the first **n** elements; **n** is a **Number** parameter
- **take(n)** - returns a new array containing the first **n** elements from the original array; **n** is a **Number** parameter
- **sum()** - returns a sum of all array elements
- **average()** - returns the average of all array elements

Input / Output

Input for functions that expect it will be passed as valid parameters. Output from functions should be their **return** value.

Constraints

Structure your code as an **IIFE**.

Hints

If we have an **instance** of and array, since we know it's an object, adding new properties to it is pretty straightforward:

```
let myArr = [1, 2, 3];

myArr.last = function () {
    // TODO
};
```

This however, only adds our new function to this instance. To add all functions just one time and have them work on **all arrays** is not much more complicated, we just have to attach them to Array's **prototype** instead:

```
Array.prototype.last = function () {
    // TODO
};
```

With such a declaration, we gain access to the context of the calling instance via **this**. We can then easily access indexes and other existing properties. Don't forget we don't want to modify the exiting array, but to create a new one:

```
Array.prototype.last = () => {
    return this[this.length - 1];
};

Array.prototype.skip = n => {
    let result = [];
    for (let i = n; i < this.length; i++) {
        result.push(this[i]);
    }

    return result;
};

Array.prototype.take = n => {
    let result = [];
    for (let i = 0; i < n; i++) {
        result.push(this[i]);
    }

    return result;
};
```

Note these functions do not have any error checking - if **n** is **negative** or **outside the bounds** of the array, and exception will be thrown, so take care when using them, or add your own validation. The last two functions require a little bit of arithmetic to be performed:

```
Array.prototype.sum = () => {
    let sum = 0;
    for (let i = 0; i < this.length; i++) {
        sum += this[i];
    }

    return sum;
};

Array.prototype.average = () => {
    return this.sum() / this.length;
};
```

To test our program in the Judge, we need to wrap it in an IIFE, like it's shown on the right. There is **no return value**, since the code execution results in functionality being added to and existing object, so they take effect instantly. We are ready to submit our solution.

```
(function solve() {
    Array.prototype.last = () => {...};

    Array.prototype.skip = n => {...};

    Array.prototype.take = n => {...};

    Array.prototype.sum = () => {...};

    Array.prototype.average = () => {...};
}());
```

2. Balloons

You have been tasked to create several classes for balloons.

Implement a class **Balloon**, which is initialized with a **color** (String) and **gasWeight** (Number). These two arguments should be **public members**.

Implement another class **PartyBalloon**, which inherits the **Balloon** class and is initialized with **2 additional parameters** - **ribbonColor** (String) and **ribbonLength** (Number).
The **PartyBalloon** class should have a **property ribbon**, which is an object with **color** and **length** - the ones given upon initialization. The ribbon property should have a **getter**.

Implement another class **BirthdayBalloon**, which inherits the **PartyBalloon** class and is initialized with **1 extra parameter** - **text** (String). The **text** should be a property and should have a **getter**.

Hints

First, we need to create a function, which will hold our classes. We create a simple function and we add the first class, the base class for all Balloons to it.

```
function solve(){
    class Baloon {
        constructor (color,gasWeight) {
            this.color = color;
            this.gasWeight = gasWeight;
        }
    }
}
```

Now that we have our base class, we can create the first child class - the **PartyBalloon**, which extends the base **Balloon** class.

Upon inheriting the **Balloon** class, the constructor of the **PartyBalloon** class will require the use of the **super()** method, to initialize the **Balloon** base constructor.

We also need to add the **ribbon** object property in the constructor of the **PartyBalloon** class. This one is for you to do.

```
function solve(){
    class Baloon {
        constructor (color,gasWeight) {
            this.color = color;
            this.gasWeight = gasWeight;
        }
    }

    class PartyBaloon extends Baloon {
        constructor(color, gasWeight, ribbonColor, ribbonLenght){
            super(color, gasWeight);
            //TODO: Initialize ribbon object
        }

        get ribbon() {
            return this._ribbon;
        }
    }

    return {
        Baloon: Baloon,
        PartyBaloon: PartyBaloon,
        BirthdayBaloon: BirthdayBaloon
    }
}
```

Now that we know how to basically inherit classes. Create the **`BirthdayBalloon`** class on your own. The **`BirthdayBalloon`** class should extend the **`PartyBalloon`** class, and should add an **extra property**. It is the same as the previous class.

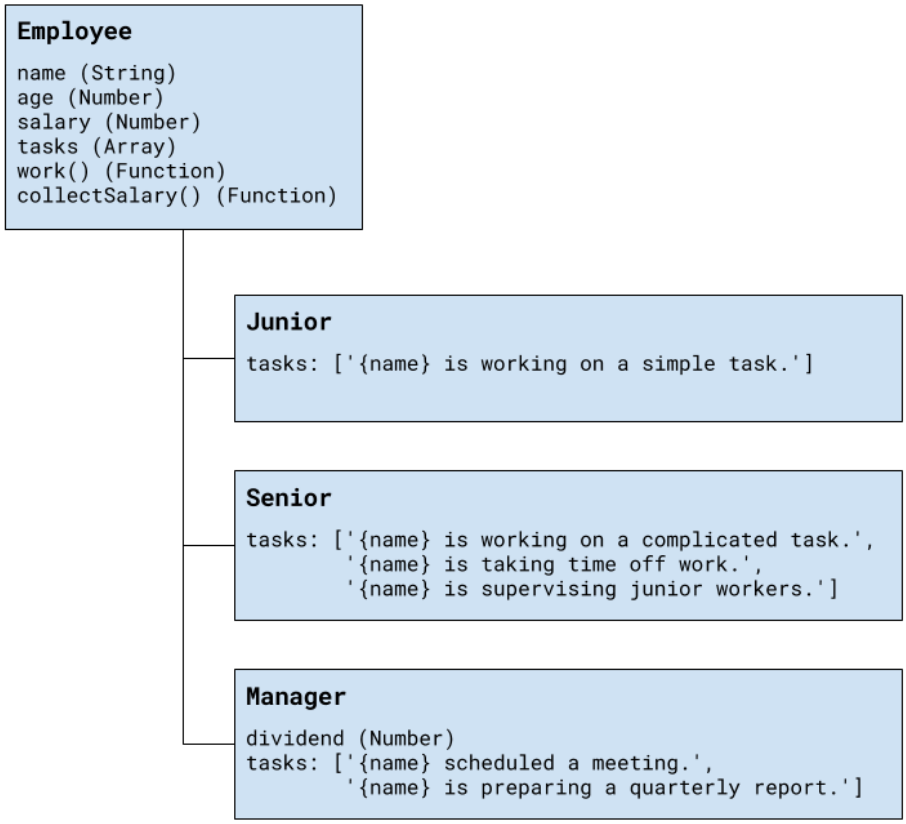
Lastly, we need to return an object, containing all of our classes, so that the Judge can work with them.

```
function solve(){
    class Baloon {...}

    class PartyBaloon extends Baloon {...}

    class BirthdayBaloon extends PartyBaloon {...}

    return {
        Baloon: Baloon,
        PartyBaloon: PartyBaloon,
        BirthdayBaloon: BirthdayBaloon
    }
}
```

Submit a **function (NOT IIFE)**, which holds all classes, and returns them as an object.

3. People

Define several 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 revealing module, containing the **three classes**. Any definitions need to be named exactly as described above.

Hints

We should begin by creating a **parent class**, that will hold all properties, shared among the different positions. Looking at the problem description, we see the following structure for out parent object:

Employee

{
 age: Number,
 name: String,
 salary: Number,
 tasks: [],
 work: Function,
 collectSalary: Function
 }

Data variables will be part of the object attached to its local context with **`this`** inside the **constructor**. Any properties that need to be initialized at instantiation time are defined as function parameters. Functions are defined inside the class body.

```
class Employee {
    constructor(name, age) {
        this.name = name;
        this.age = age;
        this.salary = 0;
        this.tasks =[];
    }

    work(){
        //TODO cycle tasks
    }

    collectSalary() {
        //TODO get paid
    }
}
```

The problem description requires that the **parent class** is **abstract**. To achieve this, we have to add a condition in the constructor which prevents its direct instantiation. Using the **`new.target`** keyword we can check whether the object was created from the abstract constructor or through a child class.

```
constructor(name, age) {
    if (new.target === Employee){
        throw  new Error("Canot instantiate directly.");
    }
    this.name = name;
    this.age = age;
    this.salary = 0;
    this.tasks =[];
}
```

The **`work()`** function has to cycle trough the list of tasks and print the current one. The easiest way to do this is to shift the first element from the array and push it at the end.

```
work(){
    let currentTask = this.tasks.shift();
    console.log(this.name + currentTask);
    this.tasks.push(currentTask);
}
```

Printing the salary is pretty straightforward. However, since the manager has an additional bonus to his salary, it's best to get the whole sum with an internal function, that the manager can **override**.

```
collectSalary() {
    console.log(`${this.name} received ${this.getSalary()} this month`);
}

getSalary(){
    return this.salary;
}
```

Now any objects that inherit from **`Employee`** will have all of its properties as well as anything new that's defined in their declaration. To inherit (extend) a class, a new class is defined with the **`extends`** keyword after its name. They also have to call the parent constructor from their own constructor, so the prototype chain is established. For **`Junior`** and **`Senior`**, the only difference from the parent **`Employee`** is the elements inside the tasks array, since they can use the functions directly from the base class. Child classes will call the parent with any parameters that are needed and push their tasks directly to the array.

```
class Junior extends Employee {
    constructor(name, age){
        super(name, age);
        this.tasks.push(' is working on simple task.');
    }
}
```

```
class Senior extends Employee {
    constructor(name, age){
        super(name, age);
        this.tasks.push(' is working on a complicated task.');
        this.tasks.push(' is taking time off work.');
        this.tasks.push(' is supervising junior workers.');
    }
}
```

The **Manager** is not much different, with the exception that his constructor has to attach a **dividend** property that is initially set to zero. His definition also needs to override the **getSalary()** function we added to the base class earlier, so it includes the bonus.

```
class Manager extends Employee {
    constructor(name, age){
        super(name, age);
        this.divident = 0;
        this.tasks.push(' scheduled a meeting.');
        this.tasks.push(' is preparing a quarterly report.');
    }

    getSalary() {
        return this.salary + this.divident;
    }
}
```

After we're done with the definitions of all object constructors, we need to wrap them in a revealing module for use by other parts of our program without polluting the global namespace, and to be submitted to the Judge:

```
function solve() {

    class Employee {...}
    class Junior extends Employee {...}
    class Senior extends Employee {...}
    class Manager extends Employee {...}

    return {Employee, Junior, Senior, Manager};
}
```

4. Posts

Your need to create several classes for **Posts**.

Implement the following classes:

- **Post**, which is initialized with **title** (String) and **content** (String)
  - The **2** arguments should be **public members**
  - The **Post** class should also have **toString()** function which returns the following result:

    ```
    "Post: {postTitle}"
    "Content: {postContent}"
    ```

- **SocialMediaPost**, which inherits the **Post** class and should be initialized with **2 additional arguments - likes** (Number) and **dislikes** (Number). The class should hold:
  - **comments(**Strings**)** - an array of strings
  - **addComment(**comment**)**- a function, which **adds** comments to that array
  - The class should extend the **toString()** function of the **Post** class, and should return the following result:

    ```
    "Post: {postTitle}"
    "Content: {postContent}"
    "Rating: {postLikes - postDislikes}"
    "Comments:"
    " * {comment1}"
    ```

```
" * {comment2}"
. . .
```

In case **there are no comments**, return information only about the **`title`**, **`content`** and **`rating`** of the **`post`**.

- **`BlogPost`**, which inherits the **`Post`** class:
  - The **`BlogPost`** class should be initialized with **1 additional argument** - **`views(`**Number**`)`**.
  - The **`BlogPost`** class should hold
    - **`view()`** - which **increments** the **`views`** of the object with **1**, every time it is called. The function should **return the object**, so that **chaining is supported**.
  - The **`BlogPost`** class should extend the **`toString()`** function of the **`Post`** class, and should return the following result:

```
"Post: {postTitle}"
"Content: {postContent}"
"Views: {postViews}"
```

Example

posts.js

```
let post = new Post("Post", "Content");

console.log(post.toString());

// Post: Post
// Content: Content

let scm = new SocialMediaPost("TestTitle", "TestContent", 25, 30);

scm.addComment("Good post");
scm.addComment("Very good post");
scm.addComment("Wow!");

console.log(scm.toString());

// Post: TestTitle
// Content: TestContent
// Rating: -5
// Comments:
//  * Good post
//  * Very good post
//  * Wow!
```

Submit a **function (NOT IIFE)**, which holds all classes, and returns them as an object.

5. *Computer

You need to implement the class hierarchy for a computer business, here are the classes you should create and support:

- **`Keyboard`** class that contains:
  - **`manufacturer`** - string property for the name of the manufacturer
  - **`responseTime`** - number property for the response time of the Keyboard
- **`Monitor`** class that contains:
  - **`manufacturer`** - string property for the name of the manufacturer
  - **`width`** - number property for the width of the screen
  - **`height`** - number property for the height of the screen

- **Battery** class that contains:
  - **manufacturer** - string property for the name of the manufacturer
  - **expectedLife** - number property for the expected years of life of the battery
- **Computer** - **abstract** class that contains:
  - **manufacturer** - string property for the name of the manufacturer
  - **processorSpeed** - a number property containing the speed of the processor in GHz
  - **ram** - a number property containing the RAM of the computer in Gigabytes
  - **hardDiskSpace** - a number property containing the hard disk space in Terabytes
- **Laptop** - class **extending** the **Computer** class that contains:
  - **weight** - a number property containing the weight of the Laptop in Kilograms
  - **color** - a string property containing the color of the Laptop
  - **battery** - an instance of the **Battery** class containing the laptop's battery. There should be a **getter** and a **setter** for the property and validation that the passed in argument is actually an instance of the Battery class.
- **Desktop** - concrete class **extending** the **Computer** class that contains:
  - **keyboard** - an instance of the **Keyboard** class containing the Desktop PC's Keyboard. There should be a **getter** and a **setter** for the property and validation that the passed in argument is actually an instance of the Keyboard class.
  - **monitor** - an instance of the **Monitor** class containing the Desktop PC's Monitor. There should be a **getter** and a **setter** for the property and validation that the passed in argument is an instance of the **Monitor** class.

Attempting to instantiate an abstract class should throw an **Error**, attempting to pass an object that is not of the expected instance (ex. an object that is not an instance of Battery to the laptop as a battery) should throw a **TypeError**.

Example

computer.js

```
function createComputerHierarchy() {
    //TODO: implement all the classes, with their properties

    return {
        Battery,
        Keyboard,
        Monitor,
        Computer,
        Laptop,
        Desktop
    }
}
```

You are asked to submit **ONLY the function** that returns an object containing the above-mentioned classes.

Bonus:

In order to achieve a better code reuse, it's a good idea to have a base abstract class containing common information - check the classes, what common characteristics do they share that can be grouped in a common base class.

6. **Mixins

Using the classes from the last task, write two mixins (functions which attach some functionality to passed in classes' prototypes) to extend their functionality. You need to support the following mixins:

- **`computerQualityMixin(classToExtend)`** - a function that attaches the following functions to the prototype of the passed in class.
    - **`getQuality()`** - returns a number equal to the computer's **`(processorSpeed + RAM + hardDiskSpace) / 3`**
    - **`isFast()`** - if **`processorSpeed > (ram / 4)`** returns **`true`**, otherwise **`false`**
    - **`isRoomy()`** - if **`hardDiskSpace > Math.floor(ram * processorSpeed)`** returns **`true`**, otherwise **`false`**
- **`styleMixin(classToExtend)`** - a function that attaches the following functions to the prototype of the passed in class:
    - **`isFullSet()`** - if the computer's **`manufacturer`**, **`keyboard's manufacturer`** and **`monitor's manufacturer`** all have the same name returns **`true`**, otherwise **`false`**.
    - **`isClassy()`** - if the computer battery's expected life is **3** years or **more** and the computer's color is either **`"Silver"`** or **`"Black"`** and the computer's weight is **less** than **3** kilograms returns **`true`**, otherwise returns **`false`**.

Examples

mixins.js

```
function createMixins() {
    // TODO: Create the mixins - computerQualityMixin and styleMixin, with all of their embedded functions
    return {
        computerQualityMixin,
        styleMixin
    }
}
```

You are asked to submit **ONLY the function** that returns an object containing the above mentioned mixins.