Problems for exercises and homework for the "JavaScript Advanced" course @ SoftUni. Submit your solutions in the SoftUni judge system at https://judge.softuni.org/Contests/2771/Prototypes-and-Inheritance-Exercise

- Array Extension

**Extend** the built-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 an array since we know it's an object, adding new properties to it is pretty straightforward:

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:

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:

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

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 an existing object, so they take effect instantly. We are ready to submit our solution.

2. String Extension

**Extend** the built-in String object with additional functionality. Implement the following functions:

- ensureStart(str) – if the current string doesn't start with the **str** parameter, return a new string in which **str** parameter **appended** to the **beginning of the current string**, otherwise returns the **original string**
- ensureEnd(str) – if the current string doesn't end with **str** parameter, return a new string in which **str** parameter **appended** to the **end of the current string**, otherwise returns the **original string**
- isEmpty() - return true if the string is empty and false otherwise
- truncate(n) – returns the string truncated to **n** characters by removing words and appends an ellipsis (three periods) to the end. If a string is less than n characters long, return the same string. If it is longer, split the string where a space occurs and append an ellipsis to it so that the total length is less than or equal to n. If no space occurs anywhere in the string, return n - 3 characters and an ellipsis. If n is less than 4, return n amount of periods.
- format(string, …params) - static method to replace placeholders with parameters. A placeholder is a number surrounded by curly braces. If parameter index cannot be found for a certain placeholder, do not modify it. Note static methods are attached to the String object instead of its prototype. See the examples for more info.

Note strings are immutable, so your functions will return new strings as a result.

Input / Output

Your **main code** should be structured as an **IIFE** without **input or output** - it should **modify** the existing String prototype instead.

Input and output of the extension functions should be as described above.

Examples

| Sample input | Value of *str* |
|---|---|
| let str = 'my string';<br>str = str.ensureStart('my');<br>str = str.ensureStart('hello ');<br>str = str.truncate(16);<br>str = str.truncate(14);<br>str = str.truncate(8);<br>str = str.truncate(4);<br>str = str.truncate(2);<br>str = String.format('The {0} {1} fox',<br>  'quick', 'brown');<br>str = String.format('jumps {0} {1}',<br>  'dog'); | 'my string'    // 'my' already present<br>'hello my string'<br>'hello my string' // length is 15<br>'hello my...'    // length is 11<br>'hello...'<br>'h...'<br>'..'<br><br>'The quick brown fox'<br><br>'jumps dog {1}'  // no parameter at 1 |

### 3. Extensible Object

Create an object that can **clone** the **functionality** of another object into itself. Implement an extend(template) function that would copy all of the properties of the **template** to the parent object and if the property is a function, add it to the object's **prototype** instead.

Input / Output

Your code should return the extensible object instance. The extend() function of your object will receive a valid object as an input parameter and has no output.

Examples

| Sample execution | Result |
|---|---|
| function extensibleObject() {<br>//TODO:<br>}<br>const myObj = extensibleObject(); | myObj: {<br>  __proto__: {}<br>  extend: function () {…}<br>} |
| const template = {<br>  extensionMethod: function () {},<br>  extensionProperty: 'someString'<br>}<br>myObj.extend(template); | myObj: {<br>  __proto__: {<br>    extensionMethod: function () {}<br>  },<br>  extend: function () {},<br>  extensionProperty: 'someString'<br>} |

Note that __proto__ is a hidden property, representing the object's prototype - depending on your test environment, you may not have access to it directly, but you can use other functions to do that.

Hints

To gain access to the prototype of an instance, use the Object.getPrototypeOf() function. To make a function shared between all instances, it'll have to be attached to the prototype instead of the instance.

### 4. 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

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

| Sample execution | Result |
|---|---|
| let classes = solution();<br>let testBalloon = new classes.Balloon("yellow", 20.5);<br>let testPartyBalloon = new classes.PartyBalloon("yellow", 20.5, "red", 10.25);<br>let ribbon = testPartyBalloon.ribbon;<br>console.log(testBalloon);<br>console.log(testPartyBalloon);<br>console.log(ribbon); | **Balloon {color: 'yellow',** gasWeight**: 20.5}**<br><br>**PartyBalloon {color: 'yellow', gasWeight: 20.5, _ribbon: {color: 'red', length: 10.25}}**<br><br>**{color: 'red', length: 10.25}** |

5. 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 below:

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 through the list of 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 **Employee**, Junior, Senior, and Manager. The property's name and age are set through 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 **four classes**. Any definitions need to be named exactly as described above.

Hints

# Example

| Sample execution | Result |
|---|---|

```
const classes = solution ();
const junior = new classes.Junior('Ivan',25);

junior.work();                          Ivan is working on a simple task.
junior.work();                          Ivan is working on a simple task.

junior.salary = 5811;                   Ivan received 5811 this month.
junior.collectSalary();

const sinior = new classes.Senior('Alex', 31);
                                        Alex is working on a complicated task.
sinior.work();                          Alex is taking time off work.
sinior.work();                          Alex is supervising junior workers.
sinior.work();                          Alex is working on a complicated task.
sinior.work();

sinior.salary = 12050;                  Alex received 12050 this month.
sinior.collectSalary();

const manager = new classes.Manager('Tom', 55);  Tom received 15000 this month.

manager.salary = 15000;                 Tom received 17500 this month.
manager.collectSalary();
manager.dividend = 2500;
manager.collectSalary();
```

6. Posts
You need to create several classes for **Posts**.
Implement the following classes:
- Post, which is initialized with the 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

<div align="center">posts.js</div>

```js
const classes = solution();
let post = new classes.Post("Post", "Content");

console.log(post.toString());

// Post: Post
// Content: Content

let scm = new classes.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.

## 7. 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 the life of the battery
- Computer – an **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 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 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.

Examples

| computer.js |
|---|
| ```js<br>function createComputerHierarchy() {<br>    //TODO: implement all the classes, with their properties<br><br>    return {<br>        Battery,<br>        Keyboard,<br>        Monitor,<br>        Computer,<br>        Laptop,<br>        Desktop<br>    }<br>}<br>``` |

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

| Sample execution | Result |
|---|---|
| ```js<br>let classes = createComputerHierarchy();<br>let Computer = classes.Computer;<br>let Laptop = classes.Laptop;<br>let Desktop = classes.Desktop;<br>let Monitor = classes.Monitor;<br>let Battery = classes.Battery;<br>let Keyboard = classes.Keyboard;<br><br>let battery = new Battery('Energy', 3);<br>console.log(battery);<br>let laptop = new Laptop("Hewlett Packard", 2.4, 4, 0.5, 3.12, "Silver", battery);<br>console.log(laptop);<br>``` | Battery {manufacturer: 'Energy', expectedLife: 3}<br><br>Laptop  {manufacturer: 'Hewlett Packard', processorSpeed: 2.4, ram: 4, hardDiskSpace: 0.5, weight: 3.12, color: 'Silver', _battery: Battery {manufacturer: 'Energy', expectedLife: 3 }} |

Bonus:

To achieve 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.