Exercise: Prototypes and Inheritance

1. 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

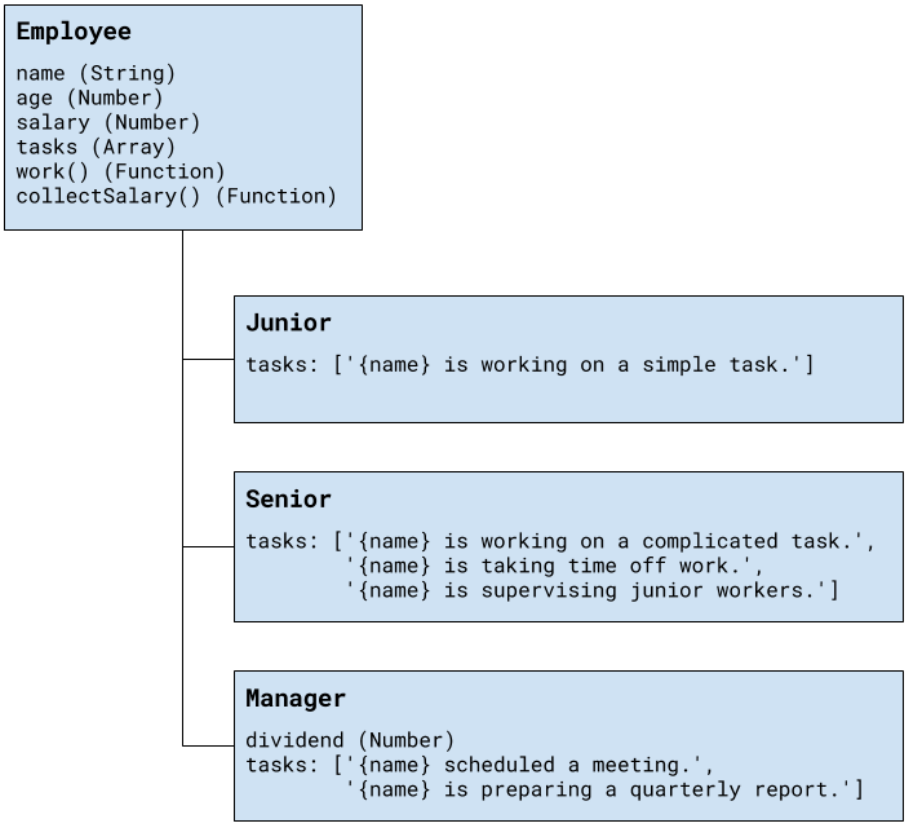
can work with them.

```
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.

2. 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};
}
```

3. 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.

4. Elemelons

If **`Watermelons`** exist, **`Firemelons`**, **`Earthmelons`** and **`Airmelons`** should also exist. Create **classes** for **Elemelons**.

Create an **abstract class** for the Elemelons. Name it **`Melon`**.

The **Melon** class should be initialized with **weight** (Number) and **melonSort** (String). The 2 arguments should be **public members**.

Create classes **Watermelon**, **Firemelon**, **Earthmelon**, **Airmelon**. Each of them should **inherit** the **abstract class Melon** and its functionality.

Aside from the abstract functionality, **each** of the **Elemelons** should have property:

- **elementIndex** (Number) - **equal** to its **weight * the string length** of its **melonSort**. The property should have only a **getter**.

All of the classes should hold a **toString()** function, which returns the following result for them:

```
"Element: {Water/Fire/Earth/Air}"
"Sort: {elemelonSort}"
"Element Index: {elemelonElementIndex}"
```

Create one more class which is called **Melolemonmelon**, which inherits **one** of the **4 elemelons**, **regardless of which**.
The **Melolemonmelon has no element**, but it can **morph** into any of the others.

- Implement a function **morph()**, which **changes the current element** of the **Melolemonmelon**, **each time** it is called
- Upon initialization, the **initial element** is **Water**. From then it should go in the following order**: Fire, Earth, Air, Water, Fire…** and so on
- The **toString()** function should remain the same as its parent class

To create an abstract class you must make sure that that class **cannot** be instantiated directly. Put the following code in the constructor of the **Melon** class, before all else.

melon.js

```
class Melon {
    constructor(weight, melonSort) {
        if (new.target === Melon) {
            throw new TypeError("Abstract class cannot be instantiated directly");
        }

        //TODO: initialize weight and melonSort properties
    }
}
```

Example

melonTests.js

```
let test = new Melon(100, "Test");
//Throws error

let watermelon = new Watermelon(12.5, "Kingsize");
console.log(watermelon.toString());

// Element: Water
// Sort: Kingsize
// Element Index: 100
```

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