# **Homework on OOP in JS**

## Task 1:

You'll need to implement 3 classes: a ***Deck***, a ***Card*** and a ***Player*** classes.

### **Deck Class Description**

#### **Class members:**

##### - **properties**:

* *cards*: contains an array of remaining cards in the deck (initially it is 52).
* *count*: number of remaining cards in the deck, should be readonly.

##### - methods:

* *shuffle()*: rearranges cards in the deck randomly.
* *draw(n)*: removes the last n cards from the deck and returns them.

#### **Implementation Details:**

Initially each deck is filled with 52 cards (13 from each of 4 suits).

### **Card Class Description:**

#### **Class members:**

##### - properties:

* *suit*: the suit of the card (i.e. ['Hearts', 'Diamonds', 'Clubs', ‘Spades']).
* *rank:* an integer from 1 to 13. ("Ace" is 1, "King" is 13, i.e. {1: 'Ace', 11: 'Jack', 12: 'Queen', 13: 'King'}).
* *isFaceCard*: a readonly property that defines whether a card is a face card (i.e. rank is either 1 or > 10).

##### - methods:

* *toString():* human-readable string representation of the card (e.g. "Ace of Spades", "10 of Clubs", "Queen of Hearts" etc.)
* *Compare(cardOne, cardTwo):* Cards must be Comparable to other cards by ranks only (no special handling for Ace).

### **Player Class Description**

#### **Class members:**

##### - properties:

* *name*: player name;
* *wins:* number of wins, readonly;
* *deck*: a deck of cards;

##### - methods:

* *Play(playerOne, playerTwo):* starts the game;

#### **Implementation Details:**

* Players both take a card from their deck.
* Whoever has a card with higher value wins the round and gets one point (if the cards are of the same value, neither of them gets a point).
* After the two cards are discarded (removed from the deck), they flip the next card from the deck.
* The game lasts until they are run out of cards.
* When game ends, figure out who is a winner (compare their scores) and return a summing up message with the final score: `{Winner} wins {X} to {Y}` (i.e. "John wins 10 to 7").



Please think carefully, what should be exposed to the world and what should be kept private (encapsulate sensitive data and unnecessary implementation details). Consider making use of static methods!

You are free to choose between ES5 and ES6 syntax. If you implement both, it will be highly appreciated.

## Task 2:

You'll need to implement inheritance in JS: a base class Employee that takes a single data object, two derived from it classes: Manager and BlueCollarWorker, and two more classes that inherit from Manager: HRManager and SalesManager.

### **Employee Class Description**

#### **Class members:**

##### - properties:

* *id*;
* *firstName*;
* *lastName*;
* *birthday*;
* *salary*;
* *position*;
* *department*;
* *age*: readonly property dynamically calculated based on birthday;
* fullName: readonly;
* *EMPLOYEES*: static readonly property where each user gets registered on initialization (contains list of all instances, except intentionally deleted once).

##### - methods:

* *quit()* - remove the employee from EMPLOYEES;
* *retire()* - log a message: "It was such a pleasure to work with you!" and remove from EMPLOYEES;
* *getFired()* - log a message: "Not a big deal!" and remove from EMPLOYEES;
* *changeDepartment(newDepartment)*;
* *changePosition(newPosition)*;
* *changeSalary(newSalary)*;
* *getPromoted(benefits)* - takes an object that can contain salary, position and department in any combination. If the property is defined, then appropriate method is called. Log: "Yoohooo!"
* *getDemoted(punishment)* - apply the same logic as to getPromoted. Log: "Damn!"

### **Manager Class Description**

#### **Implementation Details:**

inherits from Employee. Its constructor does not require position property, it should always be initialized as ‘manager’.

#### **Class members:**

##### - properties:

* *managedEmployees*: readonly, selects all instances that belong to his/her department that are not managers.

### **BlueCollarWorker Class Description**

#### **Implementation Details:**

inherits from Employee, no special logic applied here.

### **HRManager Class Description**

#### **Implementation Details:**

inherits from Manager. Its constructor does not require department property, it should always be initialized as ‘hr’.

### **SalesManager Class Implementation**

#### **Implementation Details:**

inherits from Manager. Its constructor does not require department property, it should always be initialized as ‘sales’.

## Task 3\*:

You’ll need to implement composition:

#### **Implementation Details:**

create a function **ManagerPro** that takes a manager instance and upgrades it (i.e gives him/her an ability to promote managedEmployees), you are free to add other abilities besides promotion as well.



