Ex4 specification: **Auto-Binds Decorator**

eran.amar@gmail.com

Goal

In this exercise we will understand the prototype chain (inheritance) in JS. We will distinguish between inherited properties and own properties of an object, learn how to borrow methods from the prototype and how to bind them to specific context. After you complete this exercise you will understand the differences (pro and cons) of having methods bounded to instance vs. having the methods on the prototype.

Task Overview

Write a new class that will inherit from a given class, then implement a decorator which gets a constructor function and extended it in the following way: when invoking the decorated constructor, the returned instance should *own* any inherited method (and they should be bound to that specific instance). More details in the following sections.

Files and Testers

All the files needed for that project can be found in that git repository. Download the whole repository and start fill in the missing code. You should not create new files, nor rely on any library or utils functions (even if they are being used by the testers). Write your code using vanilla JS only.
You are supplied with some sanity testers (meaning that they do **not** guarantee that your solution covered anything). To run the tests and view the result open the file `SpecRunner.html` (from the root of the folder) in your favorite browser. After changing any js file, just reload the page to rerun the testers.

Implementation Milestones and Guidance

Part 1: write a SpiderClass constructor

You are given an `AnimalClass` constructor. That class has methods and properties, and you can find the full description at: `AutoBinding\src\animalInternals.js` . Do not change that file!
You are requested to write another ctor function called `SpiderClass`.
That class should inherit from `AnimalClass` and have the following methods on its prototype:

- `spinWeb()` : increase the `numOfWebs` property that the instance owned by 1.
- `getIdSquared()` : returns the id of the instance squared (to get the id use method from `AnimalClass` prototype).

Write your code inside the file `AutoBinding\src\classes.js`

Part 2: implement bindSingleInstance function

Assume we have a class `Animal` and its prototype has a method `Animal.prototype.getName` and suppose we have an instance of that class called `dog`. We want that after applying `bindSingleInstance(dog)` the `dog` object will *own* each method from the prototype of `AnimalClass` (and from any higher parent, if any). In addition, the `this` (i.e. context) of each method should be fixed to that `dog` object. Thus, it will be safe to do the following:

```
var method = dog.getName;
method();
```

Note that without applying `bindSingleInstance`, the snippet above will throw an exception because `getName` will be missing a context.

Write your code inside the file `AutoBinding\src\bindSingleInstance.js`

Here is the function signature and document:

/**

- *obj* - an instance of arbitrary class. Might have any number of (owned) members and methods with any name. In addition might have prototype chain of any length.
- *Returns*: the same input object (after binding all inherited methods to it).

*/

```
function bindSingleInstance(obj){ … }
```

Usage example:

```
function MyClass(...) { … }
var instance = new MyClass(arg1, ...);
bindSingleInstance(instance);
var boundMethod = instance.someMethod;
boundedMethod(); // should work!
```

Part 3: implement the decorator autoBind

Implement `autoBind` decorator. That is, a function that gets a ctor-function as an input and return a new function. The returned function should behave *exactly* as the original ctor. Meaning, when calling it with `new` keyword (and proper number of arguments), it creates a new instance of that class, then apply on it the `bindSingleInstance` from previous part, and return the result.

Something to think about: Does it matter whether or not the input ctor uses the *enforcing* `new` *pattern*? Should it affect the implementation of your decorator?

Write your code inside the file `AutoBinding\src\autoBind.js`

Here is the function signature and document:

/**

- *ctorFunc* - a constructor function. Might have any number of (owned) members and methods with any name. In addition, it might need any number of arguments.
- *Returns*: new constructor function (the decorated constructor). You can assume that the decorated ctor will always be called with the `new` keyword.

*/

```
function autoBind(ctorFunc){ … }
```

Usage example:

```
function MyClass(...) { … }
var DecoratedClass = autoBind(MyClass);
var instance = new DecoratedClass(arg1, ...);
var boundMethod = instance.someMethod;
boundedMethod();  // should work!
```

Important Remarks

- It is important to link the prototypes (to make a chain) using `Object.create`. Why do you think it is useful? is this the only way to get the proper prototypes chain?
- Do not assume anything about the constructor own-enumerable properties, nor for the prototype of that class. For example, the objects you decorates might already have `hasOwnProperty` method with completely different spec. How can we overcome that? (hint)
- Simple way to goes through all inherited (and owned) properties is by:

  ```
  for (var property in obj) { do stuff... }
  ```
- We do not know in advance how many arguments the constructor needs, thus we have to invoke it with `Function.prototype.apply` , but then, how do we use the `new` keyword? Possible solution is first to `bind` the constructor to its arguments (using `apply`) and then invoke the resulting function with the `new` keyword (do that for each instance the ctor creates). Read more about it under the section “Bound Function used as constructor” at this link.
- Note that this idea of auto-binding is extremely unusual and it is not recommend using that is practice, we are practicing it only for didactic purpose.

Additional Reading and Helpful Links

- About prototypical inheritance - a must read article!
- About borrowing and binding methods, very good reading: JavaScript’s Apply, Call, and Bind Methods are Essential for JavaScript Professionals
- Private and public in JS - from crockford’s site
- Understanding the “this” in javascript: great post - please read!
- Wants to understand the testers? here is the documentation for the framework used to write the tests.

good.luck;