Assignment 1: Multiple Inheritance in Javascript

1 Introduction

This assignment is about implementing multiple inheritance in Javascript.

Attention: In this assignment you must not use the built-in new operator in Javascript because that would only make things more complicated!

2 Prototype-Based Multiple Inheritance

Javascript is a prototype based object oriented programming language. You can create new objects by calling `'Object.create(prototype)'`, where `'prototype'` becomes the prototype of the created object.

Your task is to create an object `'myObject'` with a `'create'` method that supports multiple inheritance. The method, will be called in this way:

`'myObject.create(prototypeList)'`

where `'prototypeList'` is a list of prototype objects (which may be null or empty). The `'create'` method should be reachable from all objects created by the `'create'` method.

The multiple inheritance should work as follows: When invoking a function of an object that does not exist within the object itself then it is first searched for in the receiver object's (the object that received the method call) first prototype, then in its second prototype etc. When searching for a function in a prototype, if it does not exist within the prototype then it is searched for in the prototypes of the prototype etc. Thus the search for a function should be performed in a depth-first manner and from beginning-to-end in each object's list of prototypes. The search will terminate either once a matching function is found or with an error message when the search is exhousted. [1]

To be able to control the function lookup ourselves you also need to implement a `'call(funcName, parameters)'` method, which should search for a function as described above. The `'call'` method takes two parameters: the name of the function `'funcName'` and a list of parameters `'parameters'` to the function. The `'call'` method should be implemented in `'myObject'` and should be inherited by all objects created by a call made to it, e.g. the object returned from `'myObject.create(prototypeList)'`.

Example:

```
var obj0 = myObject.create(null);
obj0.func = function(arg) { return "func0: " + arg; };

var obj1 = myObject.create([obj0]);
var obj2 = myObject.create([]);

obj2.func = function(arg) { return "func2: " + arg; };

var obj3 = myObject.create([obj1, obj2]);

var result = obj3.call("func", ["hello"])  ;
console.log("should print 'func0: hello' ->", result);
```

where `'result'` is assigned `'func0: hello`.

Another example of method lookup testing that the call method searches through all properties:

```
obj0 = myObject.create(null);
obj0.func = function(arg) { return "func0: " + arg; };

obj1 = myObject.create([obj0]);

obj2 = myObject.create([]);
```

[1]This is the simplest method resolution strategy for languages with multiple inheritance. A historically interesting language that embraced every conceivable means of combining matching methods was MIT's Flavors- see http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf

```
obj3 = myObject.create([obj2, obj1]);

result = obj3.call("func", ["hello"]);
console.log("should print 'func0: hello' ->", result);
```

Another example of method lookup testing that the call method finds methods defined in the object that is the receiver:

```
obj0 = myObject.create(null);
obj0.func = function(arg) { return "func0: " + arg; };

result = obj0.call("func", ["hello"]);
console.log("should print 'func0: hello' ->", result);
```

3 Class-Based Multiple Inheritance

Although Javascript is a prototype based object oriented programming language, you can still create particular objects, which represent classes.

Your task is to create a function for class creation that will be called in this way:

`'createClass(className, superClassList)'`

where `'className'` is the name of the created class and `'superClassList'` is a list of super-classes the created class should inherit from.

The multiple inheritance should work in a similar way as in the previous task above: When invoking a function of an object that does not exist within the object itself then it is first searched for in its class, then in the first superclass of its class, then in the second superclass of its class etc. When searching for a function in a superclass, if it does not exist within the superclass then it is searched for in the superclasses of the superclass etc. Thus the search for a function should be performed in a depth-first manner and from beginning-to-end in each class's list of superclasses. Note that since Javascript is a dynamic language new functions of an object can be added to the object without involving its class definition.

Every class object should have a function `'new()'`, which creates an instance object of the class.

Every instance object should have a method for method look-up that can be called like this:

`'call(funcName, parameters)'`

where `'funcName'` is the name of the function and `'parameters'` is a list of actual parameters that should be passed to the function. When called, the method `'call'` should perform a search for a function as described above, and invoke the function `'funcName'` with the parameters `'parameters'`.

Example:

```
var class0 = createClass("Class0", null);
class0.func = function(arg) { return "func0: " + arg; };
var class1 = createClass("Class1", [class0]);
var class2 = createClass("Class2", []);
class2.func = function(arg) { return "func2: " + arg; };
var class3 = createClass("Class3", [class1, class2]);
var obj3 = class3.new();
var result = obj3.call("func", ["hello"]);
```

where `'result'` is assigned `'func0: hello'`.

Another example of method lookup testing:

```
class0 = createClass("Class0", null);
class0.func = function(arg) { return "func0: " + arg; };

class1 = createClass("Class1", [class0]);

class2 = createClass("Class2", []);

class3 = createClass("Class3", [class2, class1]);

obj3 = class3.new();

result = obj3.call("func", ["hello"]);
```

where `'result'` is assigned `'func0: hello'`.

Another example of method lookup testing that the method will be found in the object's own class:

```
class0 = createClass("Class0", null);
class0.func = function(arg) { return "func0: " + arg; };

var obj0 = class0.new();

result = obj0.call("func", ["hello"]);
```

where `'result'` is assigned `'func0: hello'`.

4 Circular Inheritance Prevention

There is a risk for circular inheritance both with prototype-based and class-based inheritance. To get a grade A or B you must prevent such circular inheritance in your solutions of both prototype-based multiple inheritance and class-based multiple inheritance.

Example for the prototype part:

```
var obj0 = myObject.create(null);
var obj1 = myObject.create([obj0]);

obj0.addPrototype(obj1);
```

The last line above should generate an error, since it would cause circular inheritance.

Another example for the class-based part:

```
var class0 = createClass("Class 0", null);
var class1 = createClass("Class 1", [class0]);

class0.addSuperClass(class1);
```

The last line above should generate an error, since it would cause circular inheritance.

5 Handing In

The two parts of this assignment should be submitted in separate files named "part_one.js"and "part_two.js". Make sure that the name of all group members are visible both in the submission and in the code files.

6 Grading

Each part, prototype-based multiple inheritance, class-based multiple inheritance and cicular inheritance prevention, will be given a valuation with respect to the quality of the implementation:

1. NotOK means it only partly solves the problem, or that the program is substantially functioning but the code is *not* well written and structured.

2. OK means it is substantially functioning and the code is OK written and OK structured.

3. Good means it is completely functioning and the code is well written and well structured.

For the programming assignment 1 the following table will be used when grading the assignment.

Grade	Prototype-Based	Class-Based	Circular Prevention
A	Good	Good	Good
B	Good	OK	OK
C	Good	OK	NotOK
D	OK	OK	Missing/NotOK
E	OK	Missing/NotOK	Missing/NotOK
Fx	NotOK	Missing	Missing
F	Missing	Missing	Missing

Table 1: Grading scheme