Powerful JavaScript OOP concept here and now.
~~CoffeeScript, TypeScript, etc~~

JavaScript EasyOOP

Inheritance, method overriding, constructor,
anonymous classes, mixing, dynamic class extending,
packaging, static methods and variables

Zebkit
February, 2016

Simple concept

Zebkit classes has to be declared with "zebkit.Class(...)" method. The method gets the following parameters: inherited class (optionally), interfaces (optionally) and array of methods the class has to have. Class can define one constructor. Constructor is a method that doesn't have a name.

- "zebkit.Class(...)" method has to be used to declare zebkit classes
- Pass array of methods that the given class has to have
- Empty name is reserved for constructor

```
// Declare "MyClass" class
var MyClass = zebkit.Class([
    // Declare class constructor
    function () {
        this.a = 10; // Init variable "a"
    },

    // Declare class method "abc"
    function abc() {
        ...
    }
]);

// Instantiate the class
var myClassInstance = new MyClass();

// Call method "abc"
myClassInstance.abc();
myClassInstance.a; // 10
```

Access to parent context

All zebkit classes get special method "$super(...)" that has to be used to access parent class methods implementations. The big advantage of zebkit "$super(...)" usage is: it works properly for any combination of overridden class methods on every hierarchy level. Zebkit is smart enough to find a proper chain of super methods execution.

- Use $super(...) method to call a parent method implementation
- $super(...) works correct for deep hierarchy tracking properly an order of methods execution

```
var A = zebkit.Class([
    function(a) { this.varA = a; },
    function abc() { return 2; }
]);

var B = zebkit.Class(A, [
    function(a, b) {
        // Call parent constructor
        this.$super(a);
        this.varB = b;
    },
    function abc() {
        // Call parent method "abc()"
        return 3 + this.$super();
    }
]);
var a = new A(3), b = new B(2, 5);
a.varA;  // 3
b.varA;  // 2
b.varB;  // 5
b.abc(); // 3 + 2 = 5
```

Inheritance

Zebkit easyOOP supports single parent inheritance. Methods of a parent class can be overridden. According to JS concept methods cannot be overloading since number of arguments can be vary. Zebkit class constructor is also inheritable.

- Pass a parent class to be inherited as the first parameter of "Class()"
- Class methods can be overridden, but not overloaded
- Use "zebkit.instanceOf()" operator instead of standard JS "instanceof"

```
// Define class A with class method
// "abc()"
var A = zebkit.Class([
    function abc() { ... }
]);

//Declare class B that inherits class A
var B = zebkit.Class(A, [ ]);

// Instantiate the class B and A
var b = new B(), a = new A();

b.abc();  // Call inherited method
zebkit.instanceOf(b, A) // true
zebkit.instanceOf(b, B) // true
zebkit.instanceOf(a, B) // false
```

Mixing (Interface)

Mixing is way to share common functionality between classes without necessity to support improper multi inheritance OOP concept. In zebkit "mix" is an interface that can declare number of methods to be mixed with a class. Interface itself cannot inherit other interfaces and classes.

- Mixing is done with "zebkit.Interface"
- "zebkit.Interface" can declare set of methods
- Interface cannot inherit other interfaces and classes
- Interface cannot be instantiated

```
// Declare Mix1 and Mix2 interfaces
var Mix1 = zebkit.Interface([
    function abc() { ... }
]);
var Mix2 = zebkit.Interface([
    function cde() { ... }
]);

// Declare class A that mixes "Mix1"
// interface methods
var A = zebkit.Class(Mix1, [...]);

// Declare class B that mixes "Mix1"
// and "Mix2" interfaces methods
var B = zebkit.Class(Mix1,Mix2 [...]);
// Instances of A and B classes have
// got mixed methods
var a = new A(), b = new B();
a.abc();
b.abc();
b.cde();
```

Anonymous classes

Anonymous classes is one of the greatest feature zebkit EasyOOP provides. Zebkit allows developers to customize classes on the fly: during the classes instantiation you can override the class methods, constructor and extend it with new methods and variables.

- Pass as the last argument array of anonymous class methods to customize it
- Original class methods and constructor can be overridden with anonymous
- Super context is accessible from anonymous class

```
// Declare class A
var A = zebkit.Class([
    function(p) {  // Constructor
        this.v = p;
    },
    function abc() { return 10; }
]);
// Instantiate anonymous class A that
// overrides constructor and method
// “abc()”.Add new class method-“cba()”
var a = new A(1/* Constructor arg */,[
    function(p){ // Override
        this.$super(p + 1);
    },
    function abc() { // Override
        return this.$super() + 10;
    },
    function cba() { ... }
]);
a.abc(); // 20
a.v;     // 1 + 1 = 2
```

Static methods and variables

Zebkit static methods and variables have to be defined inside special "$clazz()" method that is called during class definition. The method is called in a class scope context. Static method and variables are inheritable except the variables and methods whose names start from "$" character.

- Static variables and methods have to be declared inside $clazz()
- Static variables are inheritable
- Static variables whose names start from "$" are not inheritable

```
var A = zebkit.Class([
    // Declare static stuff inside the
    // method
    function $clazz() {
        this.staticVar    = 10;
        this.$staticVar   = 11;
        this.staticMethod = function(){
            ...
        };
    }
]);

// Declare class B that inherits class A
var B = zebkit.Class(A, []);

A.staticVar;    // 10
A.$staticVar;   // 11
B.staticVar;    // 10 (inherited)
B.$staticVar;   // undefined
B.staticMethod == A.staticMethod;// true
```

Packages and namespaces

It is recommended to use package system zebkit provides. The packages are organized as hierarchy (the same way **Java** does it) and they are accessible from predefined "zebkit" global namespace using dot notation You also can define new namespaces with own packages hierarchy.

- Declare a package variables and classes safely with "zebkit.package(...)"
- A new global namespace can be declared via "zebkit.namespace(...)" method
- Use dot notation to access package stuff

```
// Define stuff in "test" package
zebkit.package("test", function() {
    this.v = "Hello 1";
});

// Access "v" from package "test"
zebkit.test.v; // "Hello 1"

//Alternative way to access the variable
zebkit("test").v; // "Hello 1"

//Define own namespace and package there
var ns = zebkit.namespace("MyNS");
ns.package("test", function() {
    this.v = "Hello 2";
});
ns.test.v; // "Hello 2"

// Alternative way to access namespace
// variables and methods
namespace("myNS").test.v; // "Hello 2"
```

Package stuff in your scope

Not always handy to access classes and variables by pointing full path to its (“zebkit.ui.grid.Grid”). Zebkit helps importing fields from the given package(s) into a local scope by using zebkit.Import(...) that should be used in combination with **JS** ”eval()” method. Import has no effect for fields whose name start from “$” character.

- Wrap zebkit code with zebkit.ready(...)
- Package fields are reachable via dot notation
- zebkit.Import() helps to import package stuff in current scope

```
// Define variables in “test” package
zebkit.package(“test”, function() {
    this.variable = “Hello 1”;
    this.A = zebkit.Class([...]);
    this.$variable = 100;
});

// Use zebkit.ready() to write
// zebkit code safely
zebkit.ready(function() {
    // Import fields from “test”
    // package into current scope
    eval(zebkit.Import(“test”));

    // Now access variables directly
    variable; // “Hello 1”
    new A();  // new instance of class
              // zebkit.test.A

    $variable;// undefined
});
```

Dynamic class extensions

Zebkit classes can be extended dynamically either on the class level (extend class definition with new methods and variables) or on the class instance level. It can be done with special "extend(...)" method. The method gets array of methods the class has to be extended as its argument.

- Use "extend(...)" method to extend a class or a class instance
- Extending of a class causes his new instances will get the extensions
- Extending an instance has effect only for the given class instance

```
var A = zebkit.Class([
    function abc() { return 10; }
]);
// Extend class "A" with "cba()" method
// and override "abc()" method
A.extend([
    function cba() { ... },
    function abc() {
        return this.$super() + 1;
    }
]);
var a = new A(), aa = new A();
a.cba(); // Call "cba()" method

// Extend instance of class "A"
a.extend([
   function q(){this.abc(); this.cba();}
]);
a.q(); // call "q()" instance specific
       // method
aa.q;  // undefined
```

Class methods via prototype

With zebkit you can define a class methods and variables following the traditional way - by adding appropriate fields to the class prototype field. Do it inside special "$prototype()" method that is called in the class "prototype" scope.

- Use "$prototype()" method to add class fields
- Don't declare complex typed variable in prototype since it is shared between all class instances
- Methods declared in prototype don't have access to super class context

```
// Declare class A
var A = zebkit.Class([
    // Old school - defining class
    // methods and variables via
    // prototype
    function $prototype() {
        // Declare variable "v"
        this.v = "Hello 1";

        // Declare class method "abc"
        this.abc = function() {
            return "Hello 2";
        };
    }
]);

var a = new A();
a.v;      // "Hello 1"
a.abc(); // "Hello 2"
```