Javascript - Última pergunta da Época Normal - 20 de Junho de 2012

```
var Ator = {
    x: 0, y:0,
    vel:0,
    init: function(x, y, vel) {
        this.x = x;
        this.y = y;
        this.vel = vel;
    }
}

var Coelho = extend(Ator, {
    cenouras: 0,
    init: function(x, y, vel) {
        Ator.init.call(this, x, y, vel, cenouras);
        this.cenouras = cenouras;
    }
})

var Cacador = extend(Ator, {
    balas: 0,
    init: function(x, y, vel, balas) {
        Ator.init.call(this, x, y, vel);
        this.balas = balas;
    }
})

var Cenoura = extend(Ator, {
    init: function(x, y) {
        Ator.init.call(this, x, y, 0);
    }
})
```

Javascript - Teste 2 - 2 Junho 2014

```
var Thermometer = {
    bottom: 0,
    top: 0,
    value: 0,
    init: function(bottom, top) {
        this.bottom = bottom;
        this.top = top;
        this.reset();    // chamar aqui o reset melhora a fatorização
    },
    get: function() {
        return this.value
    },
    set: function(value) {
        if( this.bottom <= value && value <= this.top )
            this.value = value ;
    },
    reset: function() {
        this.set((this.bottom + this.top)/2); // chamar aqui o set melhora a fa-
torização
    }
}

var MinThermometer = extend(Thermometer, {
    minValue: 99999999999.0,
// o init herdado já serve
```

```
// o get herdado já serve
    set: function(value) {
        Thermometer.set.call(this, value);
        if( this.value < this.minValue )
            this.minValue = this.value;
    },
// o reset herdado já serve
    min: function() { return this.minValue; }
})

var MultiThermometer = extend(Thermometer, {
    them: [],
    selected: 0,
    init: function(top, bottom, n) { // todos equivalentes
        Thermometer.init.class(this, bottom, top); // optional, não faz falta
mas também não faz mal
        for( var i = 0 ; i < n ; i++ )
            them.push(create(Thermometer, top, bottom));
        this.selected = 0;
    },
    get: function(value) { this.them[this.selected].get(); },
    set: function(value) { this.them[this.selected].set(value); },
    reset: function() { for( i in them ) this.them[i].reset(); },
    select: function(i) { this.selected = i; }
})

var Thermograph = extend(Thermometer, {
    history: [],
// o init herdado já serve
// o get herdado já serve
    set: function(value) {
        Thermometer.set.class(this, value);
        this.history.push(this.value);
    },
    reset: function() {
        this.history = [];
        Thermometer.reset.call(this);
    },
    average: function() {
        var s = 0;
        for( i in this.history )
            s += this.history[i];
        return s / this.history.length;
    }
})
```

Exercício 48 (JavaScript)

48 - Resolva este problema usando como orientação os dois exemplos finais da aula teórica 19. Comece por copiar as funções `create` e `extend` para o início do seu programa. Considere o problema da representação de sucessões numéricas em JavaScript. Vamos restringir-nos a três tipos de sucessões: aritméticas, geométricas e constantes.

- **Sucessão aritmética**: Caracteriza-se por um primeiro elemento a_0 e por um incremento fixo *inc*. Os elementos da sucessão são gerados usando a equação $a_i = a_{i-1}+inc$.
- **Sucessão geométrica**: Caracteriza-se por um primeiro elemento a_0 e por uma base fixa *base*. Os elementos da sucessão são gerados usando a equação $a_i=a_{i-1}$*base.

- **Sucessão constante:** Caracteriza-se pelo primeiro elemento a_0, apenas. Os elementos da sucessão constante são todos iguais ao primeiro, i.e. $a_i=a_0$.

O objetivo é que cada objeto `succ` que represente uma sucessão consiga gerar os valores dessa sucessão através da invocação repetida de `succ.next()`. Na nossa representação, as sucessões devem portanto funcionar como iteradores que se lembram da posição corrente em que vão. Atenção: não guarde os valores gerados internamente num array - isso seria desperdiçar memória inutilmente.
Cada tipo de sucessão deverá ser representado por um protótipo distinto. Os objetos criados a partir desse protótipos (usando a operação `create`), representarão sucessões concretas.
Os três protótipos devem ter a seguinte funcionalidade comum:

- int first() -- Faz reset da sucessão (isto é, coloca-a no início) e retorna o primeiro elemento a_0.
- int curr() -- Retorna o elemento corrente sem fazer avançar a sucessão.
- int next() -- Faz avançar a sucessão e retorna o novo elemento corrente.
- int at(int i) -- Retorna o i-ésimo elemento da sucessão, isto é a_i. Entretanto o elemento corrente passa a ser o i-ésimo elemento da sucessão.
- void print(int n) -- Escreve os n primeiros elementos da sucessão. Entretanto o elemento corrente passa a ser o n-ésimo elemento da sucessão.

Resolva este problema tendo em mente os seguintes objetivos:

- Minimização do código.
- Extensibilidade do sistema.

Não se preocupe com a eficiência.
Antes de programar, desenhe num papel um esquema que ilustre a relação entre os protótipos que vai definir, e ainda quais são as variáveis e os métodos de instância de cada um deles.
[Sugestões: 1. Defina uma protótipo abstrato e fatorize nele o máximo de código. 2. Note que os métodos "at" e "print" podem ser habilidosamente definidos à custa dos métodos "first" e "next".]

Uma solução possível. Repare que a função "at", definida no protótipo mais abstrato, usa a função "next", definida só nos protótipos concretos.

```
function create(proto) { // Create object and applies init(...) to it
    function F() {}
    F.prototype = proto;
    var obj = new F();
    if( arguments.length > 1 )
        obj.init.apply(obj, Array.prototype.slice.apply(arguments).slice(1));
    return obj;
}

function extend(proto, added) { // Creates new prototype that extend existing
prototype
    function F() {}
    F.prototype = proto;
    var proto1 = new F();
    for(prop in added)
        proto1[prop] = added[prop];
```

```
    return proto1;
}

var Succ = {
    a0: 0,
    current: 0,
    init: function(a0) {
        this.a0 = a0;
        this.current = a0;
    },
    first: function() {
        this.current = this.a0;
        return this.current;
    },
    curr: function() {
       return this.current;
    },
    at: function(i) {
        this.first();
        for(var j = 0 ; j < i ; j++)
            this.next();
        return this.curr();
    },
    print: function(n) {
        this.first();
        for(var j = 0 ; j < n ; j++) {
            print(this.curr());
            this.next();
        }
    }
}

var Const = extend(Succ, {
    next: function() {
        return this.curr();
    }
})

var Arith = extend(Succ, {
    inc:0,
    init: function(a0, inc) {
        Succ.init.call(this, a0);
        this.inc = inc;
    },
    next: function() {
        this.current += this.inc;
        return this.curr();
    }
})

var Geom = extend(Succ, {
    mult:0,
    init: function(a0, inc) {
        Succ.init.call(this, a0);
        this.mult = mult;
    },
    next: function() {
        this.current += this.mult;
        return this.curr();
    }
})

create(Arith,1,2).print(10);
```