PLP - Práctica 5: Programación Orientada a Objetos

Zamboni, Gianfranco

7 de marzo de 2018

Programación en JS

5.1. Ejercicio 1

a)

```
c1i = {
    r: 1,
    i: 1
}
```

b)

```
c1i.sumar = function(complejo){
    this.r += complejo.r
    this.i += complejo.i
}
```

c)

```
c1i.sumar = function(complejo){
    return {
        r: this.r + complejo.r
        i: this.i + complejo.i
    }
}
```

d) Asi como están implementadas, funciona.

e)

```
let c = c1i.sumar(c1i)

c1i.sumar = function(complejo){
    return {
        r: this.r - complejo.r
        i: this.i - complejo.i
    }
}
```

`c1i` no tiene definida la función restar, por lo que `c1i.restar(_)` nos dará un error.

f)

```
c1i.mostrar = function () {
    if(this.i != 1){
        console.log(this.r+" "+this.i+"i");
    }else{
        console.log(this.r+" "+"i");
    }
}
```

En este caso, `c` no podrá mostrar sus elementos, ya que cuando lo creamos `c1i` como un nuevo objeto que no está relacionado con `c`.

5.2. Ejercicio 2

```
// a)
let t = {}
let f = {}

t.ite = function(a,b) {
    return a;
}

f.ite = function(a,b) {
    return b;
}

// b)
t.mostrar = function() {
    return "Verdadero";
}

f.mostrar = function(){
    return "Falso";
}

// c)
t.not = function(){
    return f;
}

f.not = function(){
    return t;
}

t.and = function(otroValor) {
    return otroValor;
}

f.and = function(element) {
    return this;
}
```

5.3. Ejercicio 3

```
// a)
let Cero = {};

Cero.esCero = function(){
    return true;
};

Cero.succ = function(){
    return new Sucesor(this);
}


function Sucesor(pred){
    this.predecesor = pred;
};

Sucesor.prototype.__proto__ = Cero;

Sucesor.prototype.pred = function(){
    return this.predecesor;
}

// b)
Cero.toNumber = function(){
    return 0;
}

Sucesor.prototype.toNumber = function(){
    return this.pred().toNumber() + 1;
}

// c)
Cero.for = function(f){}

Sucesor.prototype.for = function(f){
    this.pred().for(f);
    f.eval(this);
}
```

5.4. Ejercicio 4

a)

```
var Punto = {};
Punto.new = function(x,y) {
    var p = {};
    p.x=x;
    p.y=y;
    p.mostrar = function () {
        return Punto.mostrar(this);
    }
    return p;
}

Punto.mostrar = function(o) {
    return `Punto(${o.x},${o.y})`
}
```

b)

```
var PuntoColoreado = {};
PuntoColoreado.new = function(x,y){
    var p = Punto.new(x,y);
    p.color = "rojo";
    p.mostrar = function(){
        return PuntoColoreado.mostrar(this);
    };
    return p;
};

PuntoColoreado.mostrar = function(o) {
    return Punto.mostrar(o);
};
```

c)

```
var PuntoColoreado = {};
PuntoColoreado.cons = function (x,y,color) {
    var nuevo = this.new(x,y);
    nuevo.color = color;
    return nuevo;
}
```

d)

```
\begin{lstlisting}
    var p1 = Punto.neww(1 ,2);
    var pc1 = PuntoColoreado.neww(1 ,2);
    Punto.mover = "me muevo";
    var p2 = Punto.neww(1 ,2);
    var pc2 = PuntoColoreado.neww(1 ,2);

    console.log(p1.mover);
    console.log(pc1.mover);
    console.log(p2.mover);
    console.log(pc2.mover);

  // Se pude observar que todos los objetos se pueden mover, porque
↪  los puntos coloreados tienen una referencia a los puntos, por lo
↪  tanto si se agrga una funcionalidad al punto, tanbién se
↪  agregará al punto coloreado.
```

5.5. Ejercicio 5

```
function Punto(x, y){
    this.x = x;
    this.y = y;
}

Punto.prototype.mostrar = function(){
    return `Punto(${this.x},${this.y})`
}

function PuntoColoreado(x,y, color) {
    this.x = x
    this.y = y
    this.color = color
}
PuntoColoreado.prototype.__proto__ = Punto.prototype

Punto.prototype.moverX = function(x){
    this.x += x
}
```

5.6. Ejercicio 6

a) En el primer caso, una vez creado el objeto a, el prototipo de este objeto ya queda fijo. Cuando realizamos la asignación `C1.prototype = C2.prototype;`, la variable `C1.prototype` deje de referenciar al objeto prototipo de `a` y referencie al objeto que referencia `C2.prototype`. Provocando, esto, que el constructor `C1()` lo asigne como prototipo de `b`.

Entonces `a` sigue teniendo el mismo prototipo y `b` tiene como prototipo a `C2` por lo que los resultados de evaluar sus atributos son:

```
a.g // 'Hola'
b.g // 'Mundo'
```

b) En el segundo caso, estamos modificando el atributo `g` del objeto que es referenciado por `C1.prototype`. La asignación `C1.prototype.g = C2.prototype.g;` remplaza la función `g` original de `C1.prototype` por la función `g` de `C2.prototype`. Entonces, las soluciones quedan:

```
a.g // 'Mundo'
b.g // 'Mundo'
```

5.7. Ejercicio 7

a) `a` será un array con todas las claves de `o1` y `b` será el array con todos sus valores en el mismo orden, es decir, si en `a[0]` se encuentra la clave `'a'`, entonces en `b[0]` se encuentra el valor `1`.

b)

```
function extender(o1, o2) {
    for(let key in o1) {
        if(o2[key] == undefined){
            o2[key] = o1[key];
        }
    }
}
```

c) Hay que hacer dos modificaciones: Definir en `B` el método y eliminarlo de A.

```
B.presentar = A.presentar
delete A.presentar
```

d) Es lo mismo que en el anterior, pero usando `prototype`.

```
B7d.prototype.presentar = A7d.prototype.presentar
delete A7d.prototype.presentar
```

Cálculo de Objetos

5.8. Ejercicio 8

5.8.1.

Representan al mismo objeto porque responden a los mismos mensajes. la unica variacion que se observa es el renombre del parametro this entre x,z,v.

5.8.2.

Representan a distintos objetos porque responden a los distintos mensajes.

5.9. Ejercicio 9

$o =_{def} [arg = \varsigma(x)x; val = \varsigma(x)x.arg]$.

5.9.1. o.val

$$\dfrac{\dfrac{}{o \to o}\,[\text{Obj}] \qquad \dfrac{\dfrac{}{o \to o}\,[\text{Obj}] \qquad \dfrac{}{(x)\{x \leftarrow o\} \to o}\,[\text{Obj}]}{(x.arg)\{x \leftarrow o\} \to o}\,[\text{Sel}]}{o.val \to o}\,[\text{Sel}]$$

5.9.2. o.val.arg

$$\dfrac{\dfrac{\dfrac{}{o \to o}\,[\text{Obj}]}{o.val \to o}\,[\text{Obj}] \qquad \dfrac{\dfrac{}{o \to o}\,[\text{Obj}]}{o.val \to o}\,[\text{Obj}] \qquad \dfrac{\dfrac{}{o \to o}\,[\text{Obj}] \qquad \dfrac{}{(x)\{x \leftarrow o\} \to o}\,[\text{Obj}]}{(x.arg)\{x \leftarrow o\} \to o}\,[\text{Sel}]}{(x)\{x \leftarrow o.val\} \to o}\,[\text{Sel}]}{o.val.arg \to o}\,[\text{Sel}]$$

5.9.3. $(o.arg \Leftarrow \Sigma(z)0).val$

5.10. Ejercicio 10

5.11. Ejercicio 11

5.11.1. Objetos true y false:

$true = [not : \varsigma(x)false, if : \varsigma(x)\lambda(y)\lambda(z)y, ifnot : \varsigma(x)\lambda(y)\lambda(z)z]$
$false = [not : \varsigma(x)true, if : \varsigma(x)\lambda(y)\lambda(z)z, ifnot : \varsigma(x)\lambda(y)\lambda(z)y]$

5.11.2.

$true = [not : \varsigma(x)false, if : \varsigma(x)\lambda(y)\lambda(z)y]$
$false = [not : \varsigma(x)true, if : \varsigma(x)\lambda(y)\lambda(z)z]$

5.12. Ejercicio 12

5.12.1.

$origen = [x : \varsigma(p)0, y : \varsigma(p)0, mv : \varsigma(p)\lambda(w)\lambda(z)(p.x := p.x + w).y := p.y + z]$

5.12.2. Una clase es un trait (completo) que además provee un método new. Clase Punto:

$punto =_{def} [new = \varsigma(z)[l_i = \varsigma sz.l_i(s)i \in (1..n)],\ x : \varsigma(p)0,\ y : \varsigma(p)0,\ mv : \varsigma(p)\lambda(w)\lambda(z)(p.x := p.x + w).y := p.y + z]$