Práctica N° 5 - Programación Orientada a Objetos

Programación en JS

Ejercicio 1

Se desea trabajar con números complejos.

a) Definir el objeto `c1i` que representa al número complejo $1 + i$. Este objeto tiene las propiedades r e i de tipo `number`.

b) Extender `c1i` con la operación `sumar`, que recibe como parámetro un número complejo que es sumado al receptor. Por ejemplo, `c1i.sumar(c1i); c1i.r` evalúa 2.

c) Modificar la solución anterior de manera tal que `sumar` no modifique al objeto receptor, sino que retorne un numero complejo que represente al resultado de la suma. Por ejemplo, `c1i.sumar(c1i)` evalúa `Object { r: 2, i : 2 }` pero `c1i` no ha sido modificado.

d) De acuerdo a la definición precedente de `sumar`, ¿cuál es el resultado de evaluar `c1i.sumar(c1i).sumar(c1i)`? En el caso en que el resultado sea indefinido, redefinir `c1i` de manera tal que el resultado sea `Object { r: 3, i: 3, sumar: ... }` y `c1i` no se modifique luego de la suma.

e) Definir `let c = c1i.sumar(c1i)`. Luego extender a `c` con la operación `restar` que se comporta análogamente a la definición de `sumar` en el inciso anterior. ¿Qué sucede al evaluar `c1i.restar(c)`?.

f) Extender `c1i` con una operación `mostrar` que retorna una string que representa al objeto receptor. Por ejemplo, `c1i.mostrar()` evalúa `1 + 1i`. ¿Qué sucede al evaluar `c.mostrar()`?.

Ejercicio 2 ★

Se desean definir los objetos `t` and `f` que representan respectivamente los valores de verdad verdadero y falso. Su solución no debe utilizar los tipos `boolean`, `Boolean`, `number`, ni `Number`.

a) Definir los objetos `t` y `f` que proveen sólo el método `ite` que se comporta como un condicional `if-then-else`, es decir `t.ite(a,b)` retorna `a` mientras que `f.ite(a,b)` retorna `b`.

b) Extender el protocolo de ambos objetos con la operación `mostrar`, de manera tal que `t.mostrar()` retorne `"Verdadero"` y `t.mostrar()` retorne `"Falso"`.

c) Extender a ambos objetos con las operaciones lógicas `not` y `and`.

Ejercicio 3 ★

a) Se desea definir a los números naturales como objetos. Los mismos deben proveer las operaciones `esCero` and `succ` que permiten respectivamente testear si el receptor del mensaje es 0 o no, y obtener al sucesor del receptor. Además, todos los números distintos de 0 deben proveer la operación `pred`. Ninguna de las operaciones debe modificar al objeto receptor del mensaje. Su solución no debe utilizar los tipos `number` ni `Number`.

b) Se desea agregar la operación `toNumber` que retorna el valor correspondiente de tipo `number` que representa al receptor. Por ejemplo, si `cero` es el objeto que representa a 0, la expresión `cero.succ().succ().toNumber` evalúa 2.

c) Extender la definición de naturales con el iterador `for`. Esta operación recibe por parámetro un objeto `o` que implementa el método unario `eval(n)`. Si $\underline{n}$ es el objeto que representa al número n, la expresión $\underline{n}$`.for(f)` se comporta como `o.eval(`$\underline{1}$`);...;o.eval(`$\underline{n}$`)`.

Por ejemplo,

```
let f = {eval : function (i) {console.log(i.toNumber)}};
zero.succ().succ().for(f);
```

genera la siguiente salida por consola

```
1
2
```

Ejercicio 4

Se desea implementar el siguiente modelo de clases utilizando prototipos. Su solución no debe basarse el patrón constructor y la propiedad `prototype` provista por el lenguaje.

a) Definir la clase `Punto` que provee el constructor `new(x,y)` donde `x` e `y` corresponden a las coordenadas del punto a crear. Además las instancias de la clase deben responder al mensaje `mostrar` que retorna una representación textual del punto. Por ejemplo, al evaluar `Punto.new(2,3).mostrar()` se obtiene `"Punto(2,3)"`.

Notar que la solución debe ser tal que el siguiente fragmento de código

```
let Punto = ...
let p = Punto.new(1,2);
console.log(p.mostrar());
Punto.mostrar = function(){return "unPunto"};
console.log(p.mostrar());
```

muestre por consola

```
Punto(1,2)
unPunto
```

b) Definir `PuntoColoreado` que se comporte como una subclase de `Punto` y que permita representar puntos que tienen la propiedad adicional `color`, cuyo valor inicial es `"rojo"`.

De manera análoga al caso anterior se espera que el siguiente fragmento de código

```
let Punto = ...
let PuntoColoreado = ...
let p = PuntoColoreado.new(1,2);
console.log(p.mostrar());
Punto.mostrar = function(){return "UnPunto"};
console.log(p.mostrar());
PuntoColoreado.mostrar = function(){return "UnPuntoColoreado"};
console.log(p.mostrar());
```

muestre por consola

```
Punto(1,2)
UnPunto
UnPuntoColoreado
```

c) Agregar a `PuntoColoreado` un nuevo constructor que recibe tres parámetros correspondientes a las coordenadas y al color inicial. El mismo debe reutilizar el código del constructor `new` definido previamente.

d) Considerar el siguiente fragmento de código en el que primero se definen las clases `Punto` y `PuntoColoreado` como en los incisos precedentes y luego se modifica `Punto` para proveer una nueva operación `moverX` que permite desplazar la coordenada `x` del punto en `u` unidades.

```
let Punto = ...// como en inciso a)
let PuntoColoreado = ...//como en inciso c)
let p1 = Punto.new(1,2);
let pc1 = PuntoColoreado.new(1,2);
Punto.... // Extensión de Punto para agregar moverX
let p2 = Punto.new(1,2);
let pc2 = PuntoColoreado.new(1,2);
```

Indicar cúales de los objetos `p1`, `pc1`, `p2` y `pc2` pueden responder al mensaje `moverX`. En el caso en que alguna de estas instancias no soporte la operación `moverX`, modifique su solución precente para permitir que todas puedan responder a este mensaje con el método definido en `Punto`.

Ejercicio 5 ★

Dar una solución a los incisos planteados en el ejercicio 5 (puntos y puntos coloreados) utilizando funciones constructoras y la propiedad `prototype`.

Ejercicio 6 ★

Considere el siguiente fragmento de código.

```
function C1() {};
C1.prototype.g = "Hola";

function C2() {};
C2.prototype.g = "Mundo";
```

```
let a = new C1 ();
C1.prototype = C2.prototype
let b = new C1 ();

console.log(a.g);
console.log(b.g);
```

a) Indicar qué se mostrará por consola al ejecutar dicho programa. Justificar.

b) ¿Cuál es el comportamiento del fragmento de código si se reemplaza la línea `C1.prototype = C2.prototype` por `C1.prototype.g = C2.prototype.g`?.

Ejercicio 7

a) Indicar cuál es el valor de los objetos asociados a `a` y `b` al finalizar la evaluación del siguiente fragmento de código.

```
let o = {a:1, b: function(x){return x+a}};
let o1 = Object.create(o);
o1.c = true;
let a = new Array;
let b = new Array;
for (k in o1) {a.push(k); b.push(o1[k])}
```

b) Definir una función `extender` que tome dos objetos y copie en el segundo objeto todas las propiedades del primero que no se encuentran en el segundo. Por ejemplo, `{extender ({a:1,b:true,c:"hola"},{b:1, d:"Mundo"})}` evalúa `{b:1, d:"Mundo",a:1,c:"hola"}`

c) Considere que cuenta con el siguiente fragmento de programa donde se implementa una clase `A` y una subclase `B`.

```
A = {
      inicializar: function(n,a) {this.nombre = n; this.apellido = a; this},
      presentar: function() { return this.nombre +" " + this.apellido}
      };

B = Object.create( A );
B.saludar = function() {alert( "Hola " + this.presentar() + "." )};
...
let a = Object.create(A); a.inicializar( "Juan", "Perez");
let b = Object.create(B); b.inicializar( "Pedro", "Juarez");
...
// Modificar aqui
...
```

Se debe modificar el fragmento de código de manera tal que a partir del comentario `// Modificar aqui` las "instancias" de `A` no puedan responder al mensaje `presentar` mientras que las instancias de `B` continúan utilizando la definición de `presentar` dada en la definición de `A`.

d) Indicar cómo modificaría su solución si el código fuese:

```
A  = function () {}
...
B = function () {}
...
let a = new A().inicializar( "Juan", "Perez"  );
let b = new B().inicializar( "Pedro", "Juarez" );
...
// Modificar aqui
...
```

Cálculo de Objetos

Ejercicio 8

Decir si los siguientes pares de términos definen al mismo objeto o no. Justificar

a) $o_1 \stackrel{\texttt{def}}{=} [arg = \varsigma(x)x.arg,\ val = \varsigma(x)x.arg]$ y $o_2 \stackrel{\texttt{def}}{=} [val = \varsigma(z)z.arg,\ arg = \varsigma(v)v.arg]$.

b) $o_3 \stackrel{\texttt{def}}{=} [arg = \varsigma(x)x.arg,\ val = \varsigma(x)x.arg]$ y $o_4 \stackrel{\texttt{def}}{=} [foo = \varsigma(z)z.arg,\ arg = \varsigma(v)v.arg]$.

Ejercicio 9

Considerar $o \stackrel{\texttt{def}}{=} [arg = \varsigma(x)x,\ val = \varsigma(x)x.arg]$. Derivar utilizando las reglas de la semántica operacional las reducciones para las siguiente expresiones:

a) $o.val$

b) $o.val.arg$

c) $(o.arg \Leftarrow \varsigma(z)0).val$

Ejercicio 10 ★

Sea $o \stackrel{\texttt{def}}{=} [a = \varsigma(x)(x.a := \varsigma(y)(y.a := \varsigma(z)[]))]$. Mostrar cómo reduce $o.a.a$.

Ejercicio 11 ★

a) Definir *true* y *false* como objetos con los siguientes tres métodos: *not*, *if*, and *ifnot*. Notar que tanto *if* como *ifnot* deberán retornar una función binaria. Las operaciones deberían satisfacer las siguientes igualdades:

$true.not = false$	$true.if\ v_1\ v_2 = v_1$	$false.if\ v_1\ v_2 = v_2$
$false.not = true$	$true.ifnot\ v_1\ v_2 = v_2$	$false.ifnot\ v_1\ v_2 = v_1$

b) Definir *true* y *false* como objetos que sólo proveen los métodos: *not*, *if*.

Ejercicio 12 ★

a) Definir el objeto *origen* que representa el origen de coordenadas en dos dimensiones. Este objeto provee tres operaciones: los observadores x e y y mv tal que $origen.mv\ v\ w$ desplaza a $origen$ v unidades a la derecha y w unidades hacia arriba.

b) Definir una clase *Punto*, cuyas instancias proveen las operaciones x, y y mv.

c) Mostrar como reduce *Punto.new*.

d) Definir la subclase *PuntoColoreado*, que permite construir instancias de puntos que tienen asociado un color.

Ejercicio 13

a) Considere la siguiente clase

$$plantaClass \stackrel{\texttt{def}}{=} [\quad \begin{array}{l} new = \varsigma(c)[altura = c.altura,\ crecer = \varsigma(t)c.crecer\ t], \\ altura = 10, \\ crecer = \varsigma(c)\lambda(t)t.altura := (t.altura + 10)] \end{array}$$

b) Mostrar cómo evalua *plantaClass.new.crecer*.

c) Definir *broteClass* sobreescribiendo en *plantaClass* la altura inicial por 1. La solución debería aprovechar *plantaClass* para seguir compartiendo futuras modificaciones de *plantaClass* (por ejemplo, nuevas versiones del método *crecer*).

d) Definir *malezaClass* sobreescribiendo *crecer* en *plantaClass* de manera tal que multiplique la altura de la planta por 2.

e) Escribir la clase *frutalClass* agregando a *plantaClass* el atributo *cantFrutos* inicializado en 0 y sobreescribiendo *crecer* de manera tal que se incremente la cantidad de frutos cada vez que la planta crezca.

f) Definir una función *frutalMixin* que tome una clase (por ejemplo, *plantaClass*, *broteClass*, o *malezaClass*) que retorne una nueva clase de planta frutal que se deriva de la clase dada.