

Práctica N^o 5 - Programación Orientada a Objetos

Programación en JS

Ejercicio 1

Se desea trabajar con números complejos.

- Definir el objeto `c1i` que representa al número complejo $1 + i$. Este objeto tiene las propiedades `r` e `i` de tipo `number`.
- 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.
- Modificar la solución anterior de manera tal que `sumar` no modifique al objeto receptor, sino que retorne un número 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.
- 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.
- 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)`?
- 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`.

- 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`.
- Extender el protocolo de ambos objetos con la operación `mostrar`, de manera tal que `t.mostrar()` retorne `"Verdadero"` y `f.mostrar()` retorne `"Falso"`.
- Extender a ambos objetos con las operaciones lógicas `not` y `and`.

Ejercicio 3 ★

- 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`.
- 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.
- 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 `n` es el objeto que representa al número `n`, la expresión `n.for(f)` se comporta como `o.eval(1);...;o.eval(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. No se debe usar explícitamente la propiedad `prototype` provista por el lenguaje. Notar que cuando se habla de definir una clase, el objetivo es crear un objeto que permita crear otros objetos y determinar su comportamiento, no definir una función constructora con la sintaxis provista por JavaScript a tal efecto.

- 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 cuáles 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 precedente 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

Sea el siguiente código JavaScript:

```
let c1 = {val: 1, avanzar: function(){this.val++;},
         avanzarSi: function(cond){if (cond) this.avanzar();}};
let c2 = Object.create(c1);
c2.avanzar = function(){this.val *= 2;};
c1.avanzar();
c2.avanzarSi(true);
c1.avanzarSi(true);
```

- Indicar los valores de `c1.val` y `c2.val` luego de ejecutar cada línea del código anterior (sin contar las dos líneas que definen a `c1` y `c2`). Tener en cuenta en qué momento los valores dejan de ser iguales.
- Definir una función constructora `Contador(modosDeAvance)` para crear objetos similares a `c1` y `c2`, donde `modosDeAvance` es una función que toma como parámetro el valor del atributo `val` y devuelve el valor que se asignará a este al avanzar.

Por ejemplo, si se ejecuta el siguiente código:

```
let cont = new Contador(function(v){return v+4;});
cont.avanzar();
```

entonces el valor de `cont.val` debe ser 5 al finalizar la ejecución.

Ejercicio 7 ★

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);
```

- Indicar qué se mostrará por consola al ejecutar dicho programa. Justificar.
- ¿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 8 ★

Indicar qué se muestra en la consola al ejecutar los siguientes programas:

```
let o1 = {x: 1};
let o2 = Object.create(o1);
o2.__proto__.y = 2;
console.log(o1.y);

let o1 = {x: 1};
let o2 = {y: 2};
o1.__proto__.z = 3;
console.log(o2.z);
```

Ejercicio 9

- 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 10

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

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

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

Ejercicio 11

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

a) $o.val$

b) $o.val.arg$

c) $(o.arg \leftarrow \zeta(z)o).val$

Ejercicio 12 ★

Sea $o \stackrel{\text{def}}{=} [a = \zeta(x)(x.a \leftarrow \zeta(y)(y.a \leftarrow \zeta(z)[[]]))]$. Mostrar cómo reduce $o.a.a$.

Ejercicio 13 ★

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 14 ★

- 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 15

Definir en el cálculo de objetos, el objeto *Vacio* que representa el conjunto vacío y sabe responder los siguientes mensajes:

- *hayElementos()*, que devuelve verdadero si el conjunto no contiene elementos.
- *agregar(x)*, que devuelve el objeto que agrega *x* al conjunto.
- *sacar(x)*, que devuelve el objeto que saca *x* del conjunto.
- *pertenece(x)*, que indica si *x* pertenece al conjunto.

Ejemplos:

Vacio.agregar(2).sacar(2).hayElementos() → true;

Vacio.agregar(2).agregar(3).sacar(2).pertenece(3) → true;

Ejercicio 16

- a) Considere la siguiente clase

$$plantaClass \stackrel{\text{def}}{=} [\text{new} = \zeta(c)[\text{altura} = c.\text{altura}, \text{crecer} = \zeta(t)c.\text{crecer } t], \\ \text{altura} = 10, \\ \text{crecer} = \zeta(c)\lambda(t)t.\text{altura} := (t.\text{altura} + 10)]$$

- b) Mostrar cómo evalúa *plantaClass.new.crecer*.
- c) Definir *broteClass* sobrescribiendo 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* sobrescribiendo *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 sobrescribiendo *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.