Práctica Nº 5 - Programación Orientada a Objetos

Programación en JavaScript

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` y `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. Estos deben proveer las operaciones `esCero` y `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. No se deben utilizar los tipos `number` ni `Number`.

b) Se desea agregar la operación `toNumber` que retorna el valor 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. 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 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 presente 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 anterior (puntos y puntos coloreados) utilizando funciones constructoras y la propiedad `prototype`.

Ejercicio 6

Definir la función constructora `OneTwoFunction(f)`, que recibe una función `f` de dos parámetros, y construye un objeto que puede responder el mensaje `apply` de la siguiente manera:

- Si se le pasan dos argumentos, entonces se aplica la función a dichos argumentos y se devuelve el resultado.
- Si se le pasa uno solo (es decir, si el segundo no está definido), devuelve una función de un par'ametro que, al aplicarse, devuelve el resultado de aplicar la función al argumento original y al nuevo.

Por ejemplo, al ejecutar el siguiente código:

```
let o = new OneTwoFunction((x, y) => x*y);
console.log(o.apply(1,2));
console.log(o.apply(2)(2));
```

Se deberá imprimir en consola primero 2 y luego 4.
Nota: considerar la posibilidad de modificar o extender el comportamiento de `apply` en el futuro.

Ejercicio 7

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);
```

a) 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.

b) Definir una función constructora `Contador(modoDeAvance)` para crear objetos similares a `c1` y `c2`, donde `modoDeAvance` 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 8 ★

Considere el siguiente fragmento de código.

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

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

```
let a = new C2 ();
let b = new C1 ();
let c = Object.create(a);
a = b;

console.log(a.g);
console.log(c.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 `a = b` por `C2.prototype.g = C1.prototype.g`? ¿Y si se la reemplaza por `Object.setPrototypeOf(a, b)`?

Ejercicio 9 ★

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

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

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

Ejercicio 10 ★

Sea el siguientes código escrito en JavaScript:

```
let a = { v: ()=>1, f: function(x){ return this.v + x; } };
let b = Object.create(a);
let c = { v: ()=>2 };
Object.assign(b, c);
console.log(b.f(3));
```

Indicar qué mensajes se envían durante la ejecución de la última línea, cuál es el objeto receptor de cada mensaje, y qué valor se imprime en la consola. (La suma no se considera un mensaje, ya que en JavaScript los números no son objetos).

Ejercicio 11

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) Considerar el siguiente fragmento de programa donde se implementan las funciones constructoras `A` y `B`.

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

B = function () {};
Object.setPrototypeOf(B.prototype, A.prototype);
B.prototype.saludar = function() {alert( "Hola " + this.presentar() + "." )};

let a = new A().inicializar("Juan", "Pérez");
let b = new B().inicializar("María", "Báez");

// Continuar aquí...
```

Se debe continuar el fragmento de código de manera tal que a partir del comentario `// Continuar aquí...` las "instancias" de `A` no puedan responder al mensaje `presentar` mientras que las "instancias" de `B` continúan utilizando la definición de `presentar` que usaban anteriormente las "instancias" de `A`.

Ejercicio 12

Definir el objeto `vacia` para representar una lista vacía, que puede responder a los mensajes:

- `esVacia()`: devuelve `true`.
- `cons(o)`, que retorna una lista no vacía cuya cabeza es el objeto o y como cola tiene al objeto receptor. Notar que la nueva lista debe poder responder adecuadamente a `esVacia()`, `cons(o)`, y además a `head()` y `tail()`.

Ejercicio 13

Se desea implementar un nuevo mecanismo de mensajería entre objetos. Crear el objeto `Mensajero` que se encarga de entregar mensajes (en otras palabras, de *mensajear*). La función asociada a su directiva debe recibir como argumentos un remitente, un destinatario y el mensaje que el remitente le está mandando al destinatario. El sistema de mensajería funciona de la siguiente forma:

I. Si el destinatario sabe responder al mensaje del remitente, el mensajero debe entregar el mensaje al destinatario. En caso contrario, la función retorna el mensaje original.

II. El resultado de haber entregado el mensaje al destinatario también es un mensaje que el mensajero debe entregar como respuesta del destinatario al remitente.

III. Si el remitente sabe responder a la respuesta del destinatario, el mensajero debe entregar la respuesta al remitente. En tal caso, la función debe retornar el resultado de tal ejecución. En caso contrario, la función retorna la respuesta del destinatario.

Ejercicio 14

Definir en JavaScript la función constructora `InfiniteSequence`, que genere objetos con un atributo `val` y un método `next`, con el siguiente comportamiento:

```
new InfiniteSequence().val ⇝ 1
new InfiniteSequence().next().val ⇝ 2
new InfiniteSequence().next().next().val ⇝ 3
```

...Y así sucesivamente. Es decir, el resultado de enviar el mensaje `next()` es un nuevo objeto cuyo atributo `val` es el sucesor del `val` del objeto receptor, y el resto se mantiene igual. **El objeto receptor no debe modificarse** al recibir el mensaje `next()`.

Ejercicio 15

Sea el siguiente código JavaScript:

```
function Complex(r, i) {
  this.real = r;
  this.imag = i;
}

let a {real: 1, imag: 1};
let b = Object.create(a);
b.real = 5;
let c = new Complex(2,1);
let d = Object.assign({}, c);
```

```
Complex.prototype.suma = function(otro){
  let res = Object.create(this);
  res.real += otro.real;
  res.imag = otro.imag;
  return res;
}
let e = c.suma(a);
let f = Object.assign({}, e);
Object.setPrototypeOf(f, Object.getPrototypeOf(a));
Object.setPrototypeOf(a, Complex.prototype);
Object.setPrototypeOf(b, Object.getPrototypeOf(Complex));
```

Indicar, para cada uno de los objetos a, b, c, d, e, f, cuál es su prototipo luego de ejecutar el código, y qué atributos conoce cada objeto, ya sea por tenerlos como atributos propios o por herencia. (Considerar solamente los atributos real, imag y suma.)

Ejercicio 16

Sea el siguiente código JavaScript:

```
let a = {text: "Hola"};
let b = Object.create(a);
let c = Object.assign({number: 1}, b);
let d = Object.create(c);
let e = {isWeird: true};
let f = Object.create(e);
```

```
Object.setPrototypeOf(a, Object.getPrototypeOf(d));
Object.assign(c,{prototype: b});
f.prototype = a;
Object.setPrototypeOf(e, c.prototype);
Object.setPrototypeOf(f, Object.getPrototypeOf(c));
```

Indicar las relaciones de prototipos entre los objetos a, b, c, d, e, f luego de ejecutar el código, y qué atributos conoce cada objeto (ya sea por tenerlos como atributos propios o por herencia).

Cálculo de Objetos

Ejercicio 17

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

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

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

Ejercicio 18

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

a) $o.val$

b) $o.val.arg$

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

Ejercicio 19 ★

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

Ejercicio 20 ★

a) Definir *true* y *false* como objetos con los siguientes tres métodos: *not*, *if*, e *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 *and* y *or* como objetos que se comporten como funciones que esperen dos argumentos (para poder escribir expresiones como *and(true)(false)*, etc.).

Ejercicio 21 ★

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. Considerar que los puntos se crean con sus coordenadas en 0.

c) Mostrar como reduce $Punto.new$.

d) Definir la subclase *PuntoColoreado*, que permite construir instancias de puntos que tienen asociado un color. Además del método *new*, que crea puntos blancos, la clase debe contar con un método que cree puntos de un color pasado como parámetro.

Ejercicio 22 ★

a) Se desea definir a los números naturales como objetos de manera análoga al ejercicio 3. Estos objetos deben proveer las operaciones `isZero` y `succ` que permiten respectivamente testear si el receptor del mensaje es 0 o no, y obtener al sucesor del receptor. Además, los números distintos de 0 deben proveer la operación `pred`.

b) Definir `iguales` y `menor` como objetos que se comporten como funciones que esperen dos argumentos.

Ejercicio 23

Sea el siguiente objeto:

$$\texttt{o} \stackrel{\text{def}}{=} [\ val = \varsigma(f)\lambda(n)(n.isZero).if(f.arg)((f.val(n.pred)).succ), arg = \varsigma(f)f.arg\ ]$$

a) ¿Qué representa el objeto `o`? Proponer un nombre mejor y explicar con palabras cómo funciona.

b) Considerar la regla APP vista en clase para reducir aplicaciones de forma simplificada:

$$\frac{a \longrightarrow v' \quad v' \equiv \lambda(x)b \quad b\{x \leftarrow s\} \longrightarrow v}{a(s) \approx\ v}\ [\text{APP}]$$

Mostrar cómo reduce la siguiente expresión: `o (0.succ) (0)` (Se recomienda definir objetos auxiliares.)

Ejercicio 24

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

- `hayElementos`, que devuelve `true` si el conjunto contiene al menos un elemento.
- `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` → `false`;
`Vacio.agregar(2).agregar(3).sacar(2).pertenece(3)` → `true`;
Se puede suponer que la operación `==` está definida para los elementos del conjunto.

Ejercicio 25

a) Considere la siguiente clase

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

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 *aFrutal* que tome una clase de planta (*plantaClass*, *broteClass*, o *malezaClass*) que retorne una nueva clase de planta **frutal** que se derive de la clase dada.

Ejercicio 26

a) Definir en Cálculo ς el objeto `emptyList`, que representa a la lista vacía.

Este objeto debe responder al mensaje `cons`, que recibe un elemento e y devuelve una nueva lista cuya cabeza es e y que, además de `cons`, debe poder responder a los mensajes `head` y `tail`.

b) Reducir `emptyList.cons(uno).cons(dos).tail`, indicando claramente las reglas utilizadas (siendo `uno` y `dos` los valores definidos en clase).

(Recordar que siempre se puede ponerle nombre a una expresión para no tener que escribirla varias veces.)

c) Dados los objetos `true` y `false` los definidos en el ejercicio 20, definimos el siguiente objeto:

$$\texttt{emptyListCheck} \stackrel{\text{def}}{=} [\ \texttt{isEmpty} = \texttt{true},\\ \texttt{cons} = \lambda(x)(\texttt{emptyList.cons}(x)).\texttt{isEmpty} := \texttt{false}\\ ]$$

¿Este objeto tiene el comportamiento esperado? (El comportamiento esperado es que el valor del campo `isEmpty` sea `true` si la lista es vacía y `false` en caso contrario, para cualquier lista obtenida con aplicaciones sucesivas de `cons` y `tail` a partir de `emptyListCheck`.) Justificar o indicar dónde está el problema.

Ejercicio 27

a) Definir la clase `List`, cuyo método `new` genere listas que se comporten como el objeto `emptyList` del ejercicio 26. Tener en cuenta que la clase debe tener definidos los premétodos para todos los mensajes que puedan recibir sus instancias, no solo los de la lista vacía.

b) Definir la subclase `ListCheck`, cuyas instancias posean además el atributo `isEmpty`. No se debe repetir el código de la clase `List`.

c) Comparar lo hecho en este ejercicio con el ejercicio 26, en particular los diferentes efectos de reutilizar el código ya definido en uno y otro ejercicio. ¿Qué es lo que hace que en un caso funcione mejor que en el otro?

Ejercicio 28

Sea el siguiente objeto que representa un nodo de una lista doblemente enlazada:

$$\texttt{nodo1} \stackrel{\text{def}}{=} [\ \texttt{valor} = 1,\ \texttt{anterior} = [\ ],\ \text{siguiente} = [\ ],\\ \texttt{ponerSig} = \varsigma(n)\lambda(x)n.\texttt{siguiente} := (n.\texttt{valor} := x).\texttt{anterior} \Leftarrow \varsigma(s)(n.\texttt{siguiente} := s)\ ]$$

a) Mostrar cómo reduce la expresión: `nodo1.ponerSig(2).valor`. Se puede suponer que 1 y 2 son valores.

b) Definir la clase `Nodo`, que permita definir nodos como el de arriba. Además del método `new`, que devuelve un nodo con comportamiento equivalente al de `nodo1`, debe contar con un método de clase `newWithValue`, que sea una función que espere un valor y cree un nodo con el valor recibido.