Práctica: Programación Orientada a Objetos basada en Objetos

Departamento de Computación, FCEyN, UBA

2018

JavaScript “Hola Mundo”

- Lenguaje OO basado en objetos (prototipos) imperativo.
- Un programa es un secuencia de comandos (statements), separados por “;”
  - en general “;” puede ser remplazado simplemente por el salto del línea.

```
alert("Hola");
alert("Mundo");
```

```
alert("Hola")
alert("Mundo")
```

JavaScript - Variables

- Las variables se declaran con `let` y también `var` (vieja escuela).
- Asignación `=`

```
let miVar = 1;
```

- *Case-sensitive*: `unavariable` y `unaVariable` son distintas variables.
- Se pueden declarar constantes con `const`

```
const Constante = 10;
```

JavaScript - Tipos de datos primitivos

- `number`: No hay distinción entre enteros y punto flotante. Constantes `-Infinity`, `Infinity`, `NaN`.
- `boolean`: Literales `true` y `false` y operadores `&&`, `!` y `||`.
- `string`: secuencia de 0 o mas caracteres entre comillas dobles (`"abc"`) o simples (`'a'`).
- `null`: un único valor `null` (nada, valor desconocido).
- `undefined`: un único valor `undefined` (el valor no está asignado).

El tipo de una expresión

- `typeof` es un operador que retorna una string con el nombre del tipo de la expresión.

Tipos compuestos

- Array
  - `[]`, `[1,2,true]`, `new Array()`
  - `-[-]`, `push(-)`, `pop()`, `shift()`, `unshift(_)`.
- Map
- Set
- Date
- ...

Tipado en JavaScript

- Dinámico (Tiempo de ejecución).
- Débil (conversión implícita de tipos).

```
let a = 1
typeof a // number
a += '1'
typeof a // string
```

Comparaciones

- == (mismo valor) vs === (mismo valor, mismo tipo)

```
1 == '1'//true
1 === '1'//false
0 == false//true
0 === false//false
1 == true//true
```

```
false == ""//true
false =="false"//false
null == undefined//true
null === undefined//
    false
```

- !== (distinto valor o tipo) vs != (distinto valor).

Fragmento imperativo: Flujo de control

- Condicional (la cláusula `else` puede omitirse).

```
if (cond) { ... } else { ... }
```

- Iteraciones

```
while (cond) {
  // cuerpo
}

do {
  // cuerpo
} while (cond);
```

Funciones

```
function nombre(arg1, ..., arg n) {
    // cuerpo
    // eventualmente return expr
}

let nombre = function(arg1, ..., arg n) {
    // cuerpo
}

let nombre = (arg1, arg2, ...argN) => expresion
```

Objetos

- Literal

```
let o = {a : 1, b : function (n){return 1 + n}}
// o -> Object {a : 1, b : b ()}
```

- Notación punto.

```
o.a // 1
o.b(1) // 2
```

- Parámetro `self` implícito (se escribe `this`).

```
let o = {a : 1,
         b : function (n){return this.a + n}}
```

- Redefinición (agregado)

```
o.b = function (){return this.a}
o.c = true
// o -> Object {a : 1, b : b (), c : true}
```

- Eliminación de propiedades

```
delete o.a // o -> Object {b : b (), c : true}
```

Ejercicio

Complejos (Ejercicio 1)

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.

Asignación por referencia

- Semántica de asignación por referencia.

```
let o = {b : b (), c : true};
let p = o; // p y o referencian al mismo objeto
p.d = 1; // o.d == 1
```

- == y === sobre `Object` significan 'el mismo objeto'.

Ejercicio

Complejos (Ejercicio 1)

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.

Una palabra sobre extracción de métodos

- Se pueden extraer métodos como funciones

```
let o = { a : function (n){return n+1;}};
let f = o.a;
f(0); // 1
```

- ¿Qué sucede con las referencias a `this`?

```
let o = {
  a:function(){
    return this;}
  };
o.a(); // o
```

```
let o = {
  a:function(){
    return this;}
  };
let f = o.a;
f(); // Window
```

Una palabra sobre extracción de métodos (cont)

- ¿Qué hace el siguiente fragmento?

```
let o1  = {
      f : function(){return this.g;},
      g : 1 }
let h = o1.f;
let o2 = {i : h, g : true}
o2.i ()

//evalua true
```

Ejercicio

Complejos (Ejercicio 1)

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.

Ejercicio

Complejos (Ejercicio 1), cont.

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.

Copiar propiedades

- Copiar propiedades `Object.assign(dest[, src1, ...])`

```
let o = { a: 1, b : true};
let p = {b: 2, c: 3}
let q = Object.assign({},o,p);
// q -> Object { a: 1, b: 2, c: 3};
q.a = 10;
o.a; // 1
```

- Es shallow-copy.

```
let o = { a: {b : 1}};
let p = Object.assign({},o);
// p -> Object { a: {b : 1}};
p.a.b = 10;
o.a.b; // 10
```

- (Shallow) Cloning

```
let clone = function(o){
    return Object.assign({},o);
}
```

Ejercicio

Complejos (Ejercicio 1), cont.

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)`?.

Ejercicio

Complejos (Ejercicio 1), cont.

e) 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()`?.

Prototipos y herencia

- Los objetos tienen una propiedad privada (llamada `[[Prototype]]`), cuyo valor es `null` u otro objeto, que es su prototipo.
- Notar que a propiedad `[[Prototype]]` induce una cadena
  - que finaliza con `null`.
  - no puede tener ciclos (ocurre un error en tiempo de ejecución)

Herencia de prototipo

Al intentar acceder (en lectura) a un atributo o método inexistente en un objeto, el mismo se busca en su prototipo.

Estableciendo cadena de prototipos (a)

- Utilizando `Object.setPrototypeOf(obj,prot)` y `Object.getPrototypeOf(obj)`.

```
let o1 = {a : 1}
let o2 = {b : true}
o2.a //undefined
Object.setPrototypeOf(o2,o1)
o2.a // 1
o1.b // undefined
o1.a = 2
o2.a // 2
o2.a = 3
o2.a // 3
o1.a // 2
```

- Aunque no es standard, de facto `[[Prototype]]` es implementado por el atributo `__proto__`.

```
o2.__proto__ = o1
```

`this` en una cadena de prototipos

- `this` siempre referencia al objeto receptor del mensaje, independientemente de donde se encuentra el método.

```
let o1 = {a:1, b : function (){return this.a}}
let o2 = Object.create(o1)
o2.a = 2
o2.b() // 2
```

Ejercicio

Complejos (Ejercicio 1), Ahora con prototipos

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 no modifica al receptor.

c) Definir `let c = c1i.sumar(c1i)`. Luego extender a `c` con la operación `restar`

d) Extender `c1i` con `mostrar`.

Object.create

- Crear una copia de un objeto y usar el original como prototipo es un patrón usual.
- En JavaScript, lo podemos hacer con `Object.create`.

```
let o1 = {a : 1}
let o2 = Object.create(o1)
// o2 = {a : 1}
// el prototipo de o2 es o1
// o2 --> o1
```

`Object.Prototype`

- Es el prototipo `Object`.
- Provee métodos básicos
  - `hasOwnProperty()` indica si el objeto contiene una propiedad no heredada.
  - `toString()` devuelve una string que representa al receptor.
  - métodos para establecer y buscar setters and getters.

Estableciendo cadena de prototipos (b)

- Literales.

```
let o1 = {...}
// o1 ---> Object.prototype ---> null
```

- `Object.create(prot)`

```
let o1 = {}
let o2 = Object.create(o1)
// o2 ---> o1 ---> Object.prototype ---> null

let o = Object.create(null)
// o ---> null
```

Constructores

- Se pueden crear funciones que generen objetos

```
function Punto (x,y){
  return {
         x : x,
         y: y,
         mvx : function (d){ this.x += d;}};
};
o = Punto(1,2); //Object{ x:1, y:2, mvx:mvx ()}
p = Punto(1,2); //Object{ x:1, y:2, mvx:mvx ()}
o == p // false
//o --> Object.prototype
//p --> Object.prototype
```

- Alternativamente (Funciones constructoras)

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

```
o = Punto(1,2)//Object{
    x:1, y:2, mvx:mvx()}
p = Punto(1,2)//Object{
    x:1, y:2, mvx:mvx()}
o == p //false
```

Estableciendo cadena de prototipos (c)

- Funciones constructoras.

```
function F (){}
let o = new F ()
// o ---> F.prototype --->  Object.prototype --->
    null
o.a // undefined
F.prototype.a = 1
o.a // 1
```

- Differencias entre el prototipo de `F` y los objetos creados a partir de `F`

```
// F ---> Function.prototype ---> Object.prototype
     ---> null
```

- `Function.prototype` provee las operaciones relacionadas a las funciones, por ejemplo, `apply` y `call`.

Ejercicio

Complejos (Ejercicio 1), Ahora con funciones constructoras

a) Usar el patrón de funciones constructoras para representar números complejos que proveen la operación `sumar` que no modifica al receptor.

b) Definir `let c = c1i.sumar(c1i)`. Luego extender a `c` con la operación `restar`.

c) Agregar a todos los complejos una operación `mostrar`.

d) ¿Es posible modificar su solución de manera tal que `sumar` modifique al receptor y este cambio afecte a los objetos previamente creados?

Objetos (cont.)

- Notación alternativa: mapping de string en Objeto

```
// o = {b : b (), c : true}
o["b"]  // function o.b()
o["c"] = false //o -> Object {b : b (), c : false}
```

- Chequeo de existencia de propiedad: `propiedad in obj`.

```
// o = {b : b (), c : undefined}
"a" in o // false
"c" in o // true
```

- Distinto de comparar con `undefined`.

```
//o = {b : b (), c : undefined}
o.a === undefined // true
o.c === undefined // true
```

- Iteración `for(let var in object){...}`

```
let o = {a : 1, b : 2 , c : 3};
let r = 0;
for(let p in o) { r += o[p]; };
// r = 6
```

Ejercicio

Pre- Ejercicio 7.b): `Object.assign`

a) Dar una implementación para la función `Object.assign` (que toma solo un source).