FIB

CONCEPTES AVANÇATS DE PROGRAMACIÓ
Tema 1
Col·lecció de Problemes

Recull per Jordi Delgado
(Dept. CS, UPC)

Grau d'EI, 2022-23
FIB (UPC)

1. Vam veure a classe que, si volem simular les classes tradicionals en JavaScript, à la Java o Smalltalk, ho podiem fer amb certa facilitat. La simulació es complica una mica si a més volem tenir també herència. Una manera de fer-ho és via prototipus. Si volem fer que el prototipus dels objectes instància de `B` hereti del prototipus dels objectes instància d'`A`, cal fer:

   ```
   B.prototype = Object.create(A.prototype);
   B.prototype.constructor = B;
   ```

   Què passa amb aquesta solució si mètodes heretats per les instàncies de `B` *es volen fer servir*? *Com ho podem arreglar*?

   Aquest codi us pot servir d'exemple. Volem que la classe `B` sigui subclasse de la classe `A`. Fixeu-vos en la sortida: No és el que esperaríem...

   ```
   function A() {
      this.a = 0;
      this.b = 1;
   }
   A.prototype.retornaA = function() { return this.a }
   A.prototype.retornaB = function() { return this.b }
   // provem...
   let aa = new A();
   aa.a = aa.a + 1;
   aa.b = aa.b + 1;
   console.log(aa.retornaA());
   console.log(aa.retornaB());
   function B() {
     this.a = 100;
     this.c = 101;
   }
   B.prototype = Object.create(A.prototype);  // el que hem vist a classe
   B.prototype.constructor = B;
   B.prototype.retornaC = function() { return this.c }
   // provem...
   let bb = new B();
   console.log(bb.retornaA());
   console.log(bb.retornaB());
   console.log(bb.retornaC());
   ```

2. Ja sabeu que a Javascript l'abast (*scope*) de les variables `var` és un abast de funció (*hoisting*). L'estàndar ECMAScript 6 va introduïr la possibilitat de declarar variables amb abast de bloc (utilitzant `let` en lloc de `var`), que són essencialment les que ja coneixeu i utilitzeu a Java o a C++. Expliqueu-ne la diferència i il·lustreu-ho amb un petit exemple de codi (us l'heu d'inventar). Expliqueu què és la *Temporal Dead Zone*.

3. Suposem que tenim tres funcions constructores `A`, `B` i `C`. Volem que els objectes construïts per la funció `C` puguin utilitzar les funcionalitats que proporcionen les

funcions constructores `A` i `B` (en un món OO amb classes i herència simple, diriem que `C` és una subclasse de `B` i que `B` és una subclasse d'`A`). Per exemple, si els objectes creats amb `A` tenen una propietat anomenada `propA` (de contingut inicial `a`), els objectes creats per `B` tenen una propietat anomenada `propB` (de contingut inicial `b`) i els objectes creats amb `C` tenen una propietat anomenada `propC` (de contingut inicial `c`), el resultat d'executar:

```
let c = new C();
console.log(c.propA);
console.log(c.propB);
console.log(c.propC);
```

seria:

```
a
b
c
```

4. Sabem que (quasi) tot objecte en Javascript té un prototipus (un altre objecte al que fa referència). I sabem que tot objecte-funció (objectes invocables) conté una propietat anomenada `prototype`. Aleshores, respón a aquestes qüestions:
   a) En general, el *prototipus* d'un objecte-funció i el `prototype` d'aquest objecte-funció són el mateix objecte?
   b) Hi ha cap excepció a la regla general?
   c) Per a què serveix el `prototype` d'un objecte-funció?

5. Executa aquest codi i *justifica* el resultat que obtens:

```
let temp
function f(x) {
   let temp = x
   return function () { return temp }
}
function g(x) {
   temp = x
   return function () { return temp }
}
// [a,b,c,...].map(foo) aplica foo a cada element i retorna
// [foo(a),foo(b),foo(c),...]
let qf = [1,2,3,4,5].map(f)
let qg = [1,2,3,4,5].map(g)
// [a,b,c,...].forEach(foo) aplica foo a cada element però no retorna res
// (undefined)
qf.forEach(function (e) {console.log(e())})
console.log("----")
qg.forEach(function (e) {console.log(e())})
```

6. Executa aquest codi i *justifica* el resultat que obtens, és a dir, el valor de la variable `result`:

```
function misteri(n){
   let secret = 4;
   n += 2;
   function misteri2(mult) {
      mult *= n;
      return secret * mult;
   }
   return misteri2;
}
function misteri3(param){
   function misteri4(bonus){
      return param(6) + bonus;
   }
   return misteri4;
}
let h = misteri(3);
let j = misteri3(h);
let result = j(2);
```

7.