Concepto de clausura

TSR 2018-19 Juansa Sendra

Lo que ya sabemos

- función $\longrightarrow$ computación pendiente
- Se utiliza en (casi) todos los leng. de progr.
- La declaración indica:
  - parámetros formales
  - cuerpo = expresión o grupo de sentencias

```
function min (x, y) {
    if (x<y)
        return x
    else
        return y
}
```

Lo que ya sabemos

- Puede aplicarse (invocarse) N veces, indicando en cada aplicación la lista de argumentos
  - Evalúa los argumentos, y con esos valores inicializa los parámetros formales
  - Crea un entorno local con los parámetros y las vars. locales definidas en la función
  - Ejecuta el cuerpo y devuelve un valor
  - Destruye el entorno local

```
console.log(min(4,87))  // x=4, y=87
console.log(min(2,4))   // x=2, y=4
console.log(min(2+2, min(3,27)) // x=4, y=(x=3, y=27)->3
```

¿Qué aporta la Prog. funcional?

first-class functions: La función es un tipo de datos

- Funciones anónimas (literal función) `(x) => x*2`
- Podemos asignar una función a una variable
  `const doble = (x) => x*2`
- Podemos pasar una función como argumento
  `[1,3,5,7].map(doble) // [1,6,10,14]`
- Podemos devolver una función como resultado

```
const prod = (a) => (b) => a*b
const doble = prod(2), triple = prod(3)
console.log(doble(5), triple(4)) // escribe 10 y 12
```

First-class functions

- A las variables locales de una función les podemos asignar otras funciones
  - **anidación** = definir una función dentro de otra
- **lexical scoping**: una función accede a:
  - su entorno local
  - el entorno de cada función en la que está contenida, hasta llegar al entorno global
    - variable libre: variable usada por una función pero definida fuera de su entorno local
    - **clausura** = función que accede a vars libres

First-class functions

```
const hipotenusa = (a,b) {
   const sqr = (x) => x*x  // anidacion
   const sqrt = (x) => .... // anidacion
   return sqrt( sqr(a) + sqr(b) )
}

const prod = (a) => (b) => a*b
/* en la función (b) => a*b
   la variable a es libre
   (corresponde al contexto de su función 'padre')
   Dicha funcion es una clausura
*/
```

Higher-order functions

Son aquellas funciones que reciben como argumento o devuelven como resultado otra función

- Puede estar predefinida (o definida en biblioteca)
  - `Array.map(f)` genera un nuevo array resultado de aplicar la función `f` sobre cada item del array
  - `setTimeout(f,ms)` programa la invocación de la función `f` para dentro de `ms` milisegundos
  - `emisor.on('ev', f)` asocia la invocación de la función `f` a la llegada del evento `ev` generado por `emisor`

Higher-order functions

- Y también puede definirlas el programador

```
const compose   = (a,b) => (c) => a(b(c)) // higher-order
const siguiente = (x) => x+1
const anterior  = (x) => x-1
const doble     = (x) => x*2
const mitad     = (x) => x/2

let a = [1,3,5,7]
a.map( compose(doble,anterior) )  // [0,4,8,12]
a.map( compose(mitad,siguiente) ) // [1,2,3,4]
```

Clausura (función que accede a vars libres)

- Implementación compleja 😢
  - Una clausura puede invocarse en distintos momentos
  - Debe garantizarse la existencia de los contextos anidados necesarios
    - Una función **no** destruye su contexto local cuando termina si puede requerirse para la futura activación de una clausura
    - La función inspecciona su entorno y clausura (cierra) los contextos anidados necesarios

Clausura (función que accede a vars libres)

- La complejidad de la implementación es transparente para el programador 😄
  - La resuelve el soporte para ejecución
- .. y además resulta útil para implementar 😄
  - callbacks
  - timeouts
  - proyección de argumentos
  - familias de funciones relacionadas
  - E/S asincrónica
  - Gestión de eventos

Comprobemos si ha quedado claro

- Razona cuál debería ser el resultado de evaluar esta expresión `((x) => (y) => x)(1)`
- Razona porqué esta función siempre devuelve true con independencia del argumento que se le pase `(x) => ((r1,r2) => r1 === r2)(x,x)`
- Compara los dos fragmentos de código de la transparencia siguiente, e intenta establecer el resultado final en ambos

```
const f = function () {
    for (var i=0; i<3; i++) {
        setTimeout(
            () => console.log(i),
            1000)
    }
}
f()
```

```
const f = function () {
    for (var i=0; i<3; i++) {
        ((c) => {setTimeout(
                    () => console.log(c),
                    1000)
        })(i)
    }
}
f()
```