JESUÏTES educació

DAWM06UF2

ESTRUCTURES DEFINIDES PEL PROGRAMADOR

OBJECTES

UF2.3. ESTRUCTURES DEFINIDES PEL PROGRAMADOR. FUNCIONS

CFGS Desenvolupament d'Aplicacions Web

M06. Desenvolupament web en entorn client

Fundació Jesuïtes Educació - Escola del Clot

Sergi Grau sergi.grau@fje.edu

OBJECTIUS

- Aprendre el funcionament de l'objecte Function de JS

- Aplicar les millores que aporten les darreres versions d'ES en el treball de funcions en JS

INTRODUCCIÓ

- podem utilitzar les funcions de la mateixa manera que les utilitzem en la programació clàssica, però si les assignem a una propietat, llavors tenim un mètode d'un objecte.

- per accedir al objecte dins del mètode que estem definim, utilitzem la paraula clau this.

- les funcions que s'utilitzen per a inicialitzar els objectes que estem creant, les anomenem constructors.

- en javascript les funcions son objectes, i per tant els hi podem definir propietats, i invocar altres mètodes.

- totes les definicions de funcions es poden niuar amb altres funcions, i tenen accés a qualsevol variable a l'ambit on elles estan definides, és a dir, les funcions de javascript sin tancaments (closures), i aixopossibilita tècniques de programació molt avançades.

DEFINICIÓ DE FUNCIONS

JS

```
/*utilitzem la paraula clau function.
Les funcions poden tenir identificador, o no, si les utilitzem a una
expressió en comptes d''una sentència de declaració.*/

// mostra nom i valor de cada prop d'o. Retorna undefined.
function printprops(o) {
    for (var p in o)
        console.log(p + ": " + o[p] + "\n");
}
//distància cartessiana de (x1,y1) a (x2,y2).
function distancia(x1, y1, x2, y2) {
    let dx = x2 - x1;
    let dy = y2 - y1;
    return Math.sqrt(dx * dx + dy * dy);
}
//funció recursiva
function factorial(x) {
    if (x <= 1) return 1;
    return x * factorial(x - 1);
}
```

DEFINICIÓ DE FUNCIONS

JS

```
//funció com a expressió
const quadrat = function (x) { return x * x; }
// si utilitzem noms, els podem utilitzar per a la recursivitat
const f = function fact(x) {
    if (x <= 1)
        return 1;
    else
        return x * fact(x - 1);
};
// funcions com a expressions en els arguments
dades.sort(function (a, b) { return a - b; });
//funció definida i invocada
let quadratde10 = (function (x) {
    return x * x;
}(10));
//Els noms de les funcions esdevenen variables locals.
```

CARACTERÍSTIQUES

- Les variables utilitzen programació hoisted, però les funcions(excepte les definides en expressions) també i per tant podem invocar una funció abans que aquesta estigui definida.

- La sentència return no és obligatòria, i si només tenim return sense cap expressió retorna undefined. En cas de no contenir return , la funció retorna undefined i és una acció o procediment.

- Podem niuar les funcions. Les funcions internes poden accedir a les variables més externes

```
function hipotenusa(a, b) {
  function quadrat(x) { return x*x; }
  return Math.sqrt(quadrat(a) + quadrat(b)); }
```

INVOCACIÓ

- El cos d'una funció només s'executa quan és invocada. Es poden invocar de 4 maneres:

  Com a funcions
  Com a mètodes
  Com a constructors
  Indirectament amb els mètodes call() i apply()

- Com a **funció**

```
printprops({x:1});
let total = distancia(0,0,2,1) + distancia(2,1,3,5);
```

- Com a **mètode**

```
o.m = f; // mètode m associat a una funció
//per a cridar el mètode
o.m(args);
o["m"](x,y);

var calculadora= { operand1: 1,operand2: 1,
suma: function() { this.resultat =
this.operand1 + this.operand2; }
};


calculadora.suma();
```

INVOCACIÓ

- Com a **constructor**, si una funció o mètode està precedida de l'operador new. Els constructors retornen una instància i no necessiten la paraula clau return.

```
var o = new Object();
var o = new Object;
```

- Es pot fer una invocació indirecta amb call() i apply(). Les funcions en JavaScript són objectes i tenen mètodes, com ara call i apply.

PARÀMETRES VS ARGUMENTS

- Un **paràmetre** representa un valor que el procediment espera que passi a l'cridar-ho. La declaració de procediment defineix els seus paràmetres.

- Un **argument** representa el valor que es passa a un paràmetre de procediment quan es diu a l'procediment. El codi de trucada proporciona els arguments quan crida a el procediment.

PARÀMETRES

- JavaScript no verifica el nombre de paràmetres passats a una funció, si cridem a una funció amb menys paràmetres que arguments té definits aquests valen undefined.

- Habitualment es creen funcions amb arguments opcionals, que es poden ometre. Sempre al final.

```
function obtenirNomsProps(o, /* opcional */ a)
{ if (a === undefined) a = [];
for(var prop in o) a.push(prop);
return a;  }
```

```
let a = obtenirNomsProps(o);
obtenirNomsProps(p,a);
// afegim les propietats a un array
```

- Podem verificar si el nombre de paràmetres passats és l'esperat amb l'objecte Arguments. Els arguments que manquen són undefined i els extres són ignorats.

```
function f(x, y, z) {
if (arguments.length != 3) {

throw new Error("funció cridada amb " +
arguments.length + "arguments, però espera 3");
}

}
```

PARÀMETRES PER DEFECTE

- A partir de ES2015 podem fer servir paràmetres per defecte

```
function producte(a, b = 1) {
  return a * b;
}

producte(5); // 5
```

PARÀMETRES , SINTAXI REST

- Un paràmetre REST permet representar un nombre indefinit de arguments com un array

```
function producte(multiplicador, ...arguments) {
  return arguments.map(x => multiplicador * x);
}

var arr = producte(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

FUNCTIONS FLETXA

- Les funcions fletxa són una sintaxi més curta per a expressar funcions, però no tenen this, arguments, super, etc. Sempre són anònimes i es van introduir amb ES6

```
var a = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];


var a3 = a.map(s => s.length);
console.log(a3); // logs [8, 6, 7, 9]
```

FUNCTIONS FLETXA

```
function (a, b){
  return a + b + 100;
}

// Arrow Function
(a, b) => a + b + 100;
```

```
let a = 4;
let b = 2;
function (){
  return a + b + 100;
}

let a = 4;
let b = 2;
() => a + b + 100;
```

FUNCTIONS FLETXA

```
function (a, b){
  let chuck = 42;
  return a + b + chuck;
}
```

```
// Arrow Function
(a, b) => {
  let chuck = 42;
  return a + b + chuck;
}
```

```
function bob (a){
  return a + 100;
}

// Arrow Function
let bob = a => a + 100;
```

`eval()`

The **`eval()`** method evaluates JavaScript code represented as a string.

`uneval()`

The **`uneval()`** method creates a string representation of the source code of an `Object`.

`isFinite()`

The global **`isFinite()`** function determines whether the passed value is a finite number. If needed, the parameter is first converted to a number.

`isNaN()`

The **`isNaN()`** function determines whether a value is `NaN` or not. Note: coercion inside the `isNaN` function has interesting rules; you may alternatively want to use `Number.isNaN()`, as defined in ECMAScript 2015, or you can use `typeof` to determine if the value is Not-A-Number.

`parseFloat()`

The **`parseFloat()`** function parses a string argument and returns a floating point number.

`parseInt()`

The **`parseInt()`** function parses a string argument and returns an integer of the specified radix (the base in mathematical numeral systems).

`decodeURI()`

The **`decodeURI()`** function decodes a Uniform Resource Identifier (URI) previously created by `encodeURI` or by a similar routine.

`decodeURIComponent()`

The **`decodeURIComponent()`** method decodes a Uniform Resource Identifier (URI) component previously created by `encodeURIComponent` or by a similar routine.

`encodeURI()`

The **`encodeURI()`** method encodes a Uniform Resource Identifier (URI) by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

`encodeURIComponent()`

The **`encodeURIComponent()`** method encodes a Uniform Resource Identifier (URI) component by replacing each instance of certain characters by one, two, three, or four escape sequences representing the UTF-8 encoding of the character (will only be four escape sequences for characters composed of two "surrogate" characters).

`escape()`

The deprecated **`escape()`** method computes a new string in which certain characters have been replaced by a hexadecimal escape sequence. Use `encodeURI` or `encodeURIComponent` instead.

`unescape()`

The deprecated **`unescape()`** method computes a new string in which hexadecimal escape sequences are replaced with the character that it represents. The escape sequences might be introduced by a function like `escape`. Because `unescape()` is deprecated, use `decodeURI()` or `decodeURIComponent` instead.

LES PROPIETATS CALLEE I CALLER

- En ES5 l'objecte Arguments disposa de les propietats **callee** i caller. Callee fa referència a la funció actual que s'està executant, especialment útil en el cas de funcions anònimes. **Caller** té accés a la pila de crides.

```
var factorial = function(x) {
if (x <= 1) return 1;
return x * arguments.callee(x-1); };
```

TIPUS DELS ARGUMENTS

- JavaScript és un lleguatge debilment tipat. Es important documentar les funcions per a determinar quin és el tipus dels arguments

- `function max(/* nombre… */) { /* codi*/ }`

FUNCIONS COM A VALORS

- Una de les característiques de les funcions a JavaScript és que poden ser definides en el moment de la seva invocació. Les funcions no són només sintaxi, sino que també són valors. En conseqüència es poden assignar a propietats, variables, elements d'arrays, etc.

```
function quadrat(x) { return x*x; }
//crea la funció i l'assigna a la variable quadrat

let s = quadrat; quadrat(4); // => 16
s(4); // => 16

let o = {quadrat: function(x) { return x*x; }};
let y = o.quadrat(16);

let a = [function(x) { return x*x; }, 20]; a[0]
(a[1]); // => 400
```

FUNCIONS COM A VALORS

JS

```
function suma(x, y) { return x + y; }
function resta(x, y) { return x - y; }
function producte(x, y) { return x * y; }
function divisio(x, y) { return x / y; }

function operar(operacio, operand1, operand2) {
    return operacio(operand1, operand2);
}
let i = operar(suma, operar(suma, 2, 3), operar(producte, 4, 5));
```

CREAN PROPIETATS PER A LES FUNCIONS

- Les funcions no són valors primitius a JavaScript, són un tipus **d'objecte**. Podem definit variables estàtiques associades a les funcions., com a propietats d'una funció

```
entrades.nombre= 0;
function entrades() { return entrades.nombre++; }
```

FUNCIONS COM A ESPAIS DE NOMS

- JavaScript té àmbit de visibilitat per funció i no per bloc. Les variables declarades a una funció són visibles en tota la funció, i les funcions niuades, però no existeixen fóra de la funció. Les variables declarades fóra d'una funció són globals i visibles en tot el codi JavaScript. Ens podem trobar amb problemes quan dos moduls utiltzen el mateix nom de variables. Per a solucionar-ho podem utilitzar una funció i utilitzar-la com a espai de noms.

```
function modul() {
// variables locals a la funcio
}
modul();
//idem(function() { //modul}());
```

CLOSURES (CLAUSURES)

- Les funcions a JavaScript són tancaments.

- Una cloenda és una funció que és avaluada en un entorn contenint una o més variables dependents d'un altre entorn. Quan és cridada, la funció pot accedir a aquestes variables. L'ús explícit de clausures s'associa amb programació funcional i amb llenguatges com el ML i el Lisp. Construccions com els objectes en altres llenguatges poden ser també modelats amb clausures.

- En alguns llenguatges, una cloenda pot aparèixer quan una funció està definida dins d'una altra funció, i la funció més interna refereix a les variables locals de la funció externa. En temps d'execució, quan la funció externa s'executa, es forma una clausura, consistint en el codi de la funció interna i referències a totes les variables de la funció externa que són requerides per la clausura.

CLOSURES (CLAUSURES)

- **Una clausura pot ser usada per associar una funció amb un conjunt de variables "privades", que persisteixen en les invocacions de la funció**. L'àmbit de la variable abasta només al de la funció definida a la clausura, per la qual cosa no es pot accedir per un altre codi del programa. No obstant això, la variable manté el seu valor de forma indefinida, de manera que un valor establert en una invocació roman disponible per a la següent. Lles clausures poden ser usades per amagar estats i implementar programació orientada a objectes.

- El concepte de clausura va ser desenvolupat en els anys 60 i implementat de forma completa, per primera vegada, com una característica més del llenguatge de programació Scheme. Des de llavors, molts altres llenguatges han estat dissenyats per suportar clausures.

- Àmbit lèxic o estàtic, (C) Es dedueix en temps de compilació

```
void fun()
{
    int x = 5;
    void fun2()
    {
        printf("%d", x);
    }
}
```

- cada nivell intern té accés als nivells externs.

ÀMBIT DINÀMIC

+ Àmbit dinàmic. (per exemple LISP o JavaScript)

```
void fun()
{     printf("%d", x); }

void dummy1()
{     int x = 5;      fun();}

void dummy2()
{     int x = 10;     fun();}
```

+ fun pot accedir a x a dummy1 o dummy2, o qualsevol funció que cridi a fun amb x declarada a ella

+ dummy1(); //=>5

```
var ambit= "global"; //=>global
function testAmbit() {
      var ambit = "local";
      function f() { return ambit; }
      return f();
}
testAmbit(); //=>local


//invoquem a la funció niuada, fora de la funció on
ha estat definida
//accedeix a la variable  de la funció testAmbit,
encara que cridem a f
testAmbit()(); //=> local
```

CLOSURES (CLAUSURES)

- Les funcions són executades usant la cadena d'àmbit on ha estat definida.

- Les variables locals definides en una funció externa no cesen el seu àmbit a la funció on ha estat definides.

- El funcionament és molt diferent a llenguatges com el C o assemblador.

CLOSURES (CLAUSURES)

```
// retorna una matriu de funcions que retorna 0-9
function constfuncs() {
     var funcs = [];
     for(var i = 0; i < 10; i++)
          funcs[i] = function() { return i; };
     return funcs;
}
var funcs = constfuncs();
funcs[5]() //
```

Aquest codi crea 10 closures i els desa en una matriu. Els tancament estan tots definits en la mateix invocació de la funció, i comparteixen l'accés a la variable i. Cridar a constfuncs() retorna 10, doncs la i està compartida per tots els tancaments i val 10.

LA PROPIETAT LENGTH

```
function check(args) {
var real = args.length; // nombre real d'args
var esperat = args.callee.length; // arguments
esperats
if (real !== esperat)
 throw Error("Expected " + expected + "args; got " +
actual);
}
function f(x, y, z) {
     check(arguments);
     return x + y + z;
}
```

ELS MÈTODES CALL I APPLY

```
//permeten la invocació indirecta de funcions com si
es tractés del mètode d'un altre objecte. El primer
argument és l'objecte sobre el qual la funció és
invocada, i la resta són els arguments a especificar,
en el cas de call són valors i en el cas d'apply es
passen com a arrays
f.call(o); f.apply(o);
o.m = f;
o.m();
f.call(o, 1, 2);
f.apply(o, [1,2]);
```

EL CONSTRUCTOR FUNCTION

```
//podem crear funcions a través del constructor i no
de la paraula clau function, creant funcions anònimes
//permeten la creació dinàmica de funcions
var f = new Function("x", "y", "return x*y;");
//equivalent a
var f = function(x, y) { return x*y; }
```

FUNCIONS PURES

- La programació funcional (sovint abreujada FP) és el procés de creació de programari mitjançant la composició de funcions pures, evitant l'estat compartit, dades mutables i efectes secundaris.

- Una funció que compleixi determinades condicions es considerarà com una funció pura.

- Coherència entre arguments i sortida. Els mateixos arguments sempre retornarien el mateix valor de sortida, independentment de qualsevol factor extern.

- Sense efectes secundaris.

FUNCIONS NO PURA

```
let points = 1000;
function deduct(x) {
    return points - x;
}

console.log("Points => " + deduct(5));
//Points => 995
console.log("Points => " + deduct(10));
//Points => 990

points = 500;
console.log("Points => " + deduct(5));
//Points => 495
console.log("Points => " + deduct(10));
//Points => 490
```

```
function deduct(a, x) {
    return a - x;
}

console.log("Points => " + deduct(1000, 5));
//Points => 995
console.log("Points => " + deduct(1000, 10));
//Points => 990

console.log("Points => " + deduct(500, 5));
//Points => 495
console.log("Points => " + deduct(500, 10));
//Points => 490
```

- JavaScript no és un llenguatge funcional com Lisp o Haskell, però el fet que les funcions siguin objectes facilita l'aplicació de tècniques de programació funcional amb JavaScript

```
var suma = function(x,y) { return x+y; };
var quadrat = function(x) { return x*x; };
var data = [1,1,3,5,5];
var mitj = data.reduce(suma)/data.length;
var dsv = data.map(function(x) {return x-mitj;});
var stddev = Math.sqrt(dsv.map(quadrat).reduce(suma)/
(data.length-1));
```

ITERADORS

- A JavaScript, un iterador és un objecte que defineix una seqüència i potencialment un valor de retorn després de la seva finalització.

- En concret, un iterador és qualsevol objecte que implementa el protocol Iterator mitjançant un mètode next () que retorna un objecte amb dues propietats: value i done

ITERADORS

JESUÏTES educació
DAWM06UF2 sergi.grau@fje.edu

JS

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
  let nextIndex = start;
  let iterationCount = 0;

  const rangeIterator = {
     next: function() {
         let result;
         if (nextIndex < end) {
             result = { value: nextIndex, done: false }
             nextIndex += step;
             iterationCount++;
             return result;
         }
         return { value: iterationCount, done: true }
     }
  };
  return rangeIterator;
}


const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
 result = it.next();
}

console.log( result.value);
```

FUNCIONS GENERADORES

- Les funcions del generador proporcionen una alternativa poderosa: us permeten definir un algorisme iteratiu escrivint una sola funció l'execució de la qual no és contínua. Les funcions del generador s'escriuen mitjançant la sintaxi de la funció *.

- Quan es criden, les funcions generadores no executen inicialment el seu codi. En canvi, retornen un tipus especial d'iterador, anomenat Generador. Quan es consumeix un valor trucant al següent mètode del generador, la funció Generador s'executa fins que troba la paraula clau de rendiment.

- La funció es pot cridar tantes vegades com es desitgi i retorna un generador nou cada vegada. Cada generador només es pot iterar una vegada.

FUNCIONS GENERADORES

JESUÏTES educació
DAWM06UF2 sergi.grau@fje.edu

JS

```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
  let iterationCount = 0;
  for (let i = start; i < end; i += step) {
      iterationCount++;
      yield i;
  }
  return iterationCount;
}
```

ITERABLES

+ Un objecte és iterable si defineix el seu comportament d'iteració, com ara quins valors es recopilen en un per a ... de construcció. Alguns tipus integrats, com ara Array o Map, tenen un comportament d'iteració per defecte, mentre que altres tipus (com Object) no.

JS

```
const objecteIterable = {
  *[Symbol.iterator]() {
      yield 1;
      yield 2;
      yield 3;
  }
}

for (let value of objecteIterable) {
  console.log(value);
}
// 1
// 2
// 3

o

[...objecteIterable]; // [1, 2, 3]
```

BIBLIOGRAFIA

+ https://developer.mozilla.org/ca/docs/Web/JavaScript/Reference/Functions