Profesor: Borja Rey Seoane

TEMA 4

PROGRAMACIÓNS ESTRUTURADA E ORIENTADA A OBXECTOS EN JS

Módulo: **Desenvolvemento Web en Contorna Cliente**

Ciclo: **DAW**

Programación estruturada

A programación estruturada é un paradigma de programación cuxo obxecto é mellorar a claridade e a calidade do código fonte e, simultaneamente, diminuír o tempo de desenvolvemento. Fundaméntase na emprega das estruturas de control básicas xa vistas (secuencia, selección e iteración), así como na división e organización do código necesario para resolver un problema en subrutinas ou **funcións**, que xa foron introducidas en temas anteriores e continuarán a ser obxecto de estudio neste apartado.

Hoisting

Hoisting é un concepto propio da programación que pode traducirse como alzamento ou levantamento. Consiste en que a declaración dunha variable ou función é subida até o inicio do seu ámbito. Non ocorre o mesmo coa asignación de valor ás variables, que permanece no punto do programa onde sexa realizada.

```
console.log(i); // undefined (non ten valor asignado pero xa está
declarada -mediante hoisting-)
var i=1; // Asignación de valor
console.log(i); // 1 (está declarada e ten valor)
```

O código anterior será traducido automaticamente polo intérprete de JS ao seguinte:

```
var i; // Declaración
console.log(i); // undefined
i=1; // Asignación de valor
console.log(i); // 1 (está declarada e ten valor)
```

En ámbitos máis internos (p.ex.: nunha función) acontece o mesmo:

```
console.log(j); // undefined (non ten valor asignado pero xa está
declarada -mediante hoisting-)
var j=0; // Asignación de valor
ambitoLocal();
console.log(i); // Error de referencia a variable global
```

```
function ambitoLocal() {
    console.log(i); // undefined (non ten valor asignado pero xa
está declarada –mediante hoisting-)
    var i=1; // Asignación de valor
    console.log(i); // 1 (está declarada como local e ten valor)
}
```

NOTA: ás declaracións feitas con `let` e `const` no lles é de aplicación o *hoisting*.

O *hoisting* tamén pode aplicarse a funcións, de xeito que podan ser invocadas nun punto do programa no que aínda non están declaradas.

```
faiAlgo();
function faiAlgo() {return "Ola mundo!";}
```

A excepción a esta regra ocorre coas expresións de funcións que as permiten asignar a unha variable e invocar empregando esta última:

```
var x;
x(); // Error: x is not a function
x = function () {return "Ola mundo!";}
x(); // Ola mundo!
```

NOTA: considérase unha boa práctica de programación declarar as variables ao principio dos *scripts* e funcións, o cal evita *bugs* por unha mala interpretación do funcionamento de JS (como o *hoisting*) e facilita a lexibilidade do código.

Modo estrito

Trátase dun modo do intérprete de JS que non permite o uso de variables sen declaralas coa palabra reservada `var`. Desta forma evítase a creación de variables globais por fallas na escrita, axudando a producir código máis limpo e sen erros. Outra consecuencia é que non se aplica o *hoisting* ás declaracións.

Para empregalo basta con incluír a directiva `use strict`. Ao se tratar dunha *string* e non dunha instrución, é compatible con versións de JS que non o soporten (ECMAScript 2009 ES5) posto que simplemente vana ignorar.

```
"use strict";
ambitoLocal();
console.log(i); // Punto non acadado

function ambitoLocal() {
    i=1; // Error de referencia a variable local
    console.log(i); // Punto non acadado
}
```

A directiva só é recoñecida ao principio dun *script* ou dunha función. Así, se é declarado ao principio dun *script*, terá ámbito global e todo o código será executado neste modo.

Funcións aniñadas

En JS podemos definir función dentro de outra de modo que sexa só accesible dende a función nai e as funcións irmáns (aqueloutras que foran definidas dentro da función nai, isto é, ao mesmo nivel que ela).

Do mesmo xeito que acontece co acceso ás variables, dende as funcións aniñadas tense acceso ás que estean en ámbitos superiores a elas.

```
var global = "global";
console.log(global); // global
// Acceso a variable en ámbito fillo -> ERROR
console.log(local1a); // ReferenceError
// Acceso a funcións fillas -> OK
funNivel1a();
funNivel1b();
// Acceso a función neta -> ERROR
funNivel2a(); // ReferenceError

function funNivel1a() {
    var local1a = "nivel1a";
    console.log(local1a); // nivel1a
    // Acceso a funcións fillas -> OK
    funNivel2a();
    funNivel2b();

    function funNivel2a() {
        var local2a = "nivel2a";
        console.log(local2a); // nivel2a
    }

    function funNivel2b() {
        var local2b = "nivel2b";
        console.log(local2b); // nivel2b
        // Acceso a variable en ámbito pai -> OK
        console.log(local1a); // nivel1a
        // Acceso a variable en ámbito irmán do pai -> ERROR
        console.log(local1b); // ReferenceError
        // Acceso a variable en ámbito avó -> OK
        console.log(global); // global
        // Acceso a función irmá -> OK
        funNivel2a(); // nivel2a
    }
}
```

```
function funNivel1b() {
    var local1b = "nivel1b";
    console.log(local1b); // nivel1b
    // Acceso a función filla do irmán -> ERROR
    funNivel2a(); // ReferenceError
}
```

Podemos abreviar o anterior en dúas regras sinxelas:

- Todos os elementos (variables, funcións, obxectos etc.) declarados nun mesmo ámbito son visibles entre si.
- Todos os elementos declarados en ámbitos xerarquicamente superiores (en liña directa) a un elemento son visibles para ese elemento.

Closures

Un ***closure*** é un tipo especial de obxecto que combina dous elementos: unha función e o ámbito no que foi creada a mesma. A función do *closure* terá acceso ao ámbito pai, mesmo despois de que a función nai finalice. Isto conséguese por mor da asignación da función a unha variable, que terá acceso ao ámbito tal e como esta se atopaba no intre da asignación (p.ex.: terá acceso ao valor dunha variable no momento da inicialización, sen importar se a mesma foi modificada posteriormente).

Para comprender mellor o concepto de *closure* supoñamos que queremos facer un contador de *clics* nunha páxina web. A primeira aproximación á solución podería ser a seguinte:

```
var contador = 0;
function incrementar() { contador++; }
incrementar();
incrementar();
incrementar(); // contador = 3 (se non é modificado por outro
código)
```

O problema que se propón con esta solución é que calquera código da páxina poderá modificar o contador (por se tratar dunha variable global). Polo tanto, o contador debería ser accesible só dende a función que o modifica:

```
function incrementar() {
  var contador = 0;
  return ++contador;
}
incrementar();
incrementar();
incrementar(); // contador = 1 (o contador inicialízase a 0 cada
vez)
```

Neste caso o problema que aparece é que o contador torna a cero a cada volta que é accedida a función que o incrementa. Unha primeira idea para resolvelo é recorrer ás funcións aniñadas, mediante algo semellante a isto:

```
function engadir() {
  var contador = 0;
  function incrementar() { contador++; }
  return contador;
}
incrementar();
incrementar();
incrementar(); // ReferenceError (función non accesible)
```

Pero a función que incrementa o contador non é accesible dende fora. A solución definitiva conséguese coa utilización dun *closure*:

```
var incrementar = (function () {
  var contador = 0;
  return function () { return ++contador; }
})();
incrementar();
incrementar();
incrementar(); // contador = 3 (só accesible a través da función)
```

A función anónima autoinvocada que é asignada á variable `incrementar` só é executada unha vez. Por iso é idónea para iniciar o contador que, ademais, pode considerarse unha variable privada que só é accedida dende a función anónima que a modifica. Tamén é a encargada de asignar á variable `incrementar` a expresión de función para realizar o acceso ao contador no ámbito pai.

No seguinte exemplo, análogo funcionalmente ao anterior, podemos velo un chisco máis claro, separando a función orixinal que crea o contador:

```
// Esta función devolve outra función encargada de incrementar o
contador.
function crearContador() {
    var contador = 0;
    return function () {
        contador++;
        console.log(contador);
        return contador;
    }
}

// Ao gardar o resultado de executar a función "crearContador"
nunha variable estamos a crear un closure que conterá a función
devolta e máis o ámbito da función "crearContador" (iso é o que
permite que non se "perda" a variable contador)
var incrementar = crearContador();
```

```
incrementar();
incrementar();
incrementar();
```

Programación Orientada a Obxectos en JS

A POO (Programación Orientada a Obxectos) é un paradigma de programación baseado no concepto do obxecto en tanto contedor de datos e posuidor dunha serie de accións que traballan sobre eses datos, xunta coa capacidade de se comunicar con outros obxectos. Con esta forma de encapsulación conséguese que non sexa preciso coñecer os detalles de implementación interna para traballar cun obxecto, o que redunda na creación de programas de funcionalidade máis complexa.

As linguaxes OO teñen mecanismos distintos para soportar o manexo de obxectos, pero as máis populares son as baseadas en clases. Os obxectos son, polo tanto, instancias en memoria das clases, as cales determinan o tipo dos primeiros (isto é, as súas características xerais e comportamento).

Neste sentido, aínda que JS non é unha linguaxe orientada a obxectos en sentido estrito (aínda que si baseada en obxectos), permite programar OO xa sexa definindo directamente os obxectos ou mediante o traballo con clases (a partir de ECMAScript 2015 ES6).

Os obxectos e clases supoñen unha capa de abstracción a maiores ao tempo de deseñar aplicacións, posto que permiten abordar a resolución de problemas prestando menos atención aos detalles de implementación interna e máis á creación de entidades e a interacción entre elas.

Obxectos

Os obxectos son entidades cargadas en memoria que conteñen datos (atributos ou propiedades) e funcións ou procedementos que traballan con eles (métodos), de xeito que todos os obxectos dun mesmo tipo (p.ex.: unha persoa) teñen as mesmas propiedades (p.ex.: nome, idade, etc.) pero distintos valores. Así mesmo, terán os mesmos métodos, pero cada obxecto executaraos cos seus propios valores e en intres distintos do seu ciclo de vida.

O obxectos facilitan exclusivamente unha interface ao exterior para traballar con eles, sen acceder directamente a elementos internos cuxos detalles de implementación non teñen por que ser coñecidos.

Como as variables, trátase de contedores de datos, pero almacenan valores compostos no canto de simples. En JS un obxecto é unha colección de pares `nome:valor`, nos que os valores asignados poden ser variables simples, funcións ou outros obxectos. Pódense declarar directamente de forma literal, indicando os pares `nome:valor` separados por comas (`,`) e entre chaves (`{}`):

```
// Variable simple
var persoa = "Xiao";

// Obxecto
var persoa = {nome:"Xiao", idade:23, peso: 78};
```

Outra forma de facer o mesmo sería:

```
// Obxecto
var persoa = new Object();
persoa.nome = "Xiao";
persoa.idade = 23;
persoa.peso = 78;
```

Acceso

Para acceder aos atributos e métodos dos obxectos existen dúas formas:

- **Notación de punto (`.`):** cuxa sintaxe é `<obxecto>.<propiedade>`.
- **Indexando o nome da propiedade (`[]`):** cuxa sintaxe aproximada `<obxecto>["<propiedade>"]`.

```
persoa.nome; // Xiao
persoa["nome"]; // Xiao
```

Arrays asociativos

Xa que JS implementa os *arrays* con indexación de base cero, o acceso indexado a obxectos permite simular o que se coñece coma *arrays* asociativos, nos que os índices poden ser etiquetas para facilitar e darlle máis significado ao acceso aos elementos:

```
var temperaturas = { "primavera":15, "veran":25, "outono":20,
"inverno":10 };
```

O acceso aos elementos terase que facer empregando as etiquetas. Se se utiliza a indexación numérica dos *arrays* nativos, obteremos un valor `undefined`:

```
console.log(temperaturas["inverno"]); // 10
console.log(temperaturas[0]); // undefined
```

Por ese motivo non se pode empregar un bucle for habitual para recorrelos:

```
for (var i=0; i< temperaturas.length; i++) { //
temperaturas.length == 0
  console.log(temperaturas[0]); // undefined
}
```

Para percorrer un *array* asociativo hai que empregar a variante `for-in` (non funcionaría o `for-of`):

```
for (var i in temperaturas) {
  console.log(temperaturas[i]);
}

for (var i of temperaturas) { // TypeError -> temperaturas not
iterable
  console.log(i);
}
```

Métodos

Os métodos son accións que se poden levar a cabo cos obxectos, normalmente para acceder ou modificar os seus atributos. Isto conséguese gracias a que se almacenan nas propiedades do obxecto como definicións de funcións e, polo tanto, forman parte do contexto do mesmo.

```
var persoa = {
  nome:"Xiao", idade:23, altura: 181, peso: 78,
  // Métodos
  facerAnos: function () { return ++this.idade; },
  calcularIMC: function () { return this.peso /
(this.altura*this.altura/10000); }
};
persoa.facerAnos(); // 24
persoa.calcularIMC(); // 23.8088
```

Esta conexión entre métodos e atributos é un dos eixos centrais da orientación a obxectos.

this

Dentro dun método, o obxecto especial `this` refírese ao propietario do mesmo. Polo tanto no exemplo anterior, equivale ao obxecto `persoa` (ou ao obxecto `Global` nunha función normal) e, consecuentemente, `this.idade` refírese á propiedade `idade` do obxecto `persoa`.

Notas acerca do valor de `this` en outros contextos:

- Dentro dunha función en `strict mode` devolverá un valor `undefined`.
- Fóra dunha función referirase ao obxecto `global` (p.ex.: `window` nunha contorna web).
- Na resposta a un evento referirase ao elemento que xerou o evento.

Un comportamento particular de `this` prodúcese cando invocamos un método dun obxecto a través das funcións predefinidas `call()` ou `apply()`. Neste caso o obxecto `this` pode referirse a outro obxecto pasado como parámetro:

```
var persoa = {
  nome:"Xiao", idade:23, altura: 181, peso: 78,
  // Métodos
  facerAnos: function () { return ++this.idade; },
  calcularIMC: function () { return this.peso /
(this.altura*this.altura/10000); }
};
var persoa2 = { nome:"Antía", idade:24 }
persoa.facerAnos.call(persoa2); // 25
```

Clases

As clases fornecen unha definición do tipo que terán os obxectos. Con elas non se pode traballar directamente, senón que deberán ser instanciadas para crear obxectos de xeito que sexan estes os que desenvolvan a lóxica do programa. Esta definición das clases constará das propiedades (datos) e métodos (accións) de que disporán os obxectos.

As clases en JS son un tipo de función que permite crear prototipos de obxectos, podéndose definir de dúas formas, segundo empreguemos as palabras clave `function` ou `class`:

- **`function`:** o primeiro método de creación de prototipos proposto por JS baseábase na emprega de funcións.

```
function Persoa(nome) {
  this.nome = nome; // this.nome refírese ao atributo da clase
}
```

```
// Creación dun obxecto 'autor' da clase 'Persoa'
var autor = new Persoa("Xiao Castro");
```

- **`class`:** o segundo método (aparecido coa versión ECMAScript 2015 ES6) emprega a palabra clave `class`, e permite asignar as propiedades dentro dun método `constructor()`, o cal será chamado automáticamente cada vez que se instancie un obxecto.

```
class Persoa {
  constructor(nome) {
       this.nome = nome; // this.nome refírese ao atributo da
clase
  }
}

// Creación dun obxecto 'autor' da clase 'Persoa'
var autor = new Persoa("Xiao Castro");
```

Se non se define o método `construtor`, JS creará un automáticamente, invisible e baleiro.

A sintaxe da definición de clases debe estar en modo estrito, o que significa que todas as variables deben ser declaradas para non obter erros:

```
class Persoa {
  constructor(nome) {
    i = 0; // ERRO de sintaxe -> declarar con var/let/const
    this.nome = nome;
  }
}
var autor = new Persoa("Xiao Castro");
```

Métodos

Ademais do método `construtor` obrigatorio, pódense engadir outros personalizados que accedan ás propiedades da clase a través palabra clave `this` (tanto se son creados mediante `function` coma se o facemos mediante `class`).

```
class Persoa {
  constructor(nome) {
    this.nome = nome;
  }

  presentarA(x) {
    return "Ola, " +x+ "! O meu nome é " +this.nome;
  }
}
```

```
// Creación dun obxecto 'autor' da clase 'Persoa'
var autor = new Persoa("Xiao Castro");
autor.presentarA("Don Pepito"); // Ola, Don Pepito! O meu nome é
Xiao Castro
```

Un tipo particular son os métodos estáticos, que son definidos na propia clase e non no seu prototipo. Isto implica que non se pode invocar desde os obxectos instanciados en base á clase, senón desde a propia clase:

```
class Persoa {
  constructor(nome) {
    this.nome = nome;
  }

  static presentar() { return "Ola!"; }
  static presentarme(obxecto) { return "Ola! O meu nome é "
+obxecto.nome; }
}

var autor = new Persoa("Xiao Castro");
console.log(autor.presentar()); // ERRO: Non se pode acceder ao
método estático
Persoa.presentar(); // Ola!
```

Se queremos acceder aos atributos dun obxecto desde un método estático, deberá ser pasado como parámetro:

```
var autor = new Persoa("Xiao Castro");
Persoa.presentarme(autor); // Ola! O meu nome é Xiao Castro
```

Herdanza

A herdanza é un mecanismo de reutilización de código mediante o cal poden crearse novas clases formando unha estrutura xerárquica en base á reutilización de atributos e métodos de clases base xa existentes.

Empregando a palabra clave `extends`, a clase nova herda todos os métodos da existente.

Dentro da definición dunha clase na que se emprega herdanza o obxecto especial `super` fai referencia á clase pai. Polo tanto, invocando o método `super()` no construtor da nova clase, execútase o da clase pai para ter acceso ás propiedades desta última.

```
class Persoa {
  constructor(nome) { this.nome = nome; }
  static presentar() { return "Ola!"; }
  presentarme() { return "Ola! O meu nome é " +this.nome; }
  presentarA(x) { return "Ola, " +x+ "! O meu nome é " +this.nome;
  }
}

class Docente extends Persoa {
  constructor(nome, departamento) {
    super(nome);
    this.departamento = departamento;
  }
  presentarTraballo() { return this.presentarme() +" e traballo en
"+ this.departamento; }
}

var administrador = new Docente("Saínza", "Informática");
administrador.presentarTraballo(); // Ola! O meu nome é Saínza e
traballo en Informática
```

Getters/Setters

Na POO é habitual fornecer métodos para ler (coñecidos coma *getters*) ou modificar (coñecidos coma *setters*) os valores dos atributos dos obxectos, de xeito que non se acceda directamente a estes últimos. Isto é especialmente útil por dous motivos:

- Independiza a utilización dun obxecto a través da súa interface (o conxunto dos seus métodos) da representación interna que se empregue para os datos (o conxunto dos seus atributos). Desta forma, se queremos cambiar os atributos dun obxecto, só haberá que modificar en consecuencia os métodos que acceden a eles, pero non as instrucións da aplicación nas que outros obxectos empregan o obxecto que modificamos.
- Permite facer algún pre-tratamento dos datos antes de devolvelos ou modificalos.

Para engadir estes métodos hai que empregar as palabras clave `get` e `set` seguidas dun nome que represente o atributo ao que acceden (non poden ser iguais). Unha práctica habitual é antepoñer un guión baixo (_) ao nome dos atributos e empregar o mesmo nome sen el para os *getters*/*setters*:

```
class Persoa {
  constructor(nome) { this._nome = nome; }
  get nome() { return this._nome; } // Getter
  set nome(x) { this._nome = x; } // Setter
}

// Creación dun obxecto 'autor' da clase 'Persoa'
var autor = new Persoa("Xiao");
autor.nome = "Xiao Castro"; // Setter <- Xiao Castro
console.log(autor.nome); // Getter -> Xiao Castro
```

NOTA: aínda que os *getters*/*setters* son métodos, non se empregan parénteses (`()`) para invocalos, senón que a súa invocación emprega a mesma forma que o acceso directo a atributos.

Hoisting

O mecanismo de *hoisting* non aplica á declaración das clases do mesmo xeito que aplicaba a variables e funcións, xa que obrigadamente esta debe aparecer denantes instanciar os obxectos:

```
// Non se pode empregar a clase denantes definila
var autor = new Persoa("Xiao"); // ReferenceError

class Persoa {
  constructor(nome) { this._nome = nome; }
}

// Creación dun obxecto 'autor' da clase 'Persoa'
var autor = new Persoa("Xiao");
```

JSON

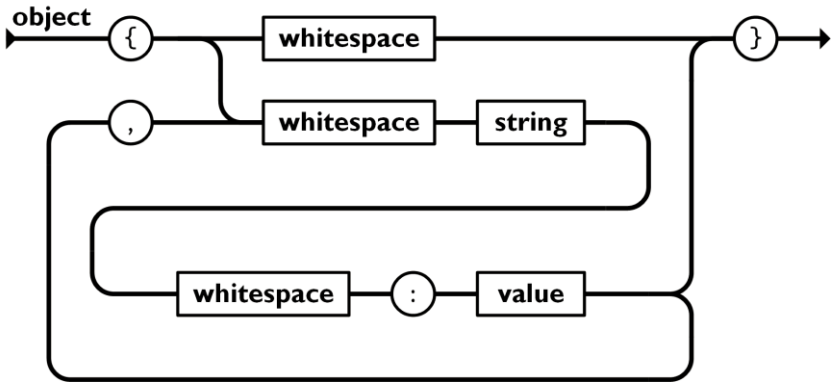
JSON[1] é un formato de intercambio de datos lixeiro baseado na estrutura dos obxectos JS (aínda que independente da linguaxe empregada para manexalo). É doado de entender para os usuarios humanos e simple de procesar para as computadoras, en tanto que ten unha estrutura semellante á das linguaxes de marcas.

Estrutura

A organización da información en JSON baséase en dúas estruturas:

- Unha colección de pares nome/valor (ou clave/valor, dependendo do autor consultado), que equivalen aos atributos/propiedades dun obxecto xunta cos seus valores asociados.
- Unha listaxe ordenada de valores que pode representarse a nivel de código coma un *array* ou outro tipo de estrutura de datos.

No seguinte diagrama podemos ver a sintaxe dun JSON xenérico:



Así mesmo os valores que aparecen indicados no diagrama anterior poden posuír calquera dos tipos recoñecidos por JS (`string`, `number`, `object`, `array`, `true`, `false` ou `null`).

[1] JavaScript Object Notation.

Velaquí un exemplo dun JSON no que se almacenan tres obxectos de tipo persoa como elementos dun array que é, ao seu tempo, o valor dun atributo doutro obxecto:

```
{
"empregados":[
  {"nome":"Brais", "apelidos":"Doce Castro"},
  {"nome":"Uxía", "apelidos":"Pereira Seivane"},
  {"nome":"Saínza", "apelidos":"Carballo Louro"}
]
}
```

Manipulación

Os JSON poden ser procesados no código JS coma se foran obxectos ou *strings* e, para iso, apoiámonos no uso de dous métodos incorporados na API JSON, sendo estes:

- **`parse()`:** recibe un JSON en forma de texto e devolve un obxecto (ou un *array* se o que recibe é a representación en *string* dun deles). Por exemplo:

```
let stringEmpregados = '{ "empregados":[
  {"nome":"Brais", "apelidos":"Doce Castro"},
  {"nome":"Uxía", "apelidos":"Pereira Seivane"},
  {"nome":"Saínza", "apelidos":"Carballo Louro" }]}';
const obxectoEmpregados = JSON.parse(stringEmpregados);
```

- **stringify():** recibe un obxecto e devolve a *string* JSON correspondente ao mesmo. Velaquí un exemplo:

```
const stringEmpregados = JSON.stringify(obxectoEmpregados);
```

Restricións

Para poder cumprir o estándar de JSON, é preciso que as estruturas que escribamos respecten unha serie de limitacións:

- Os nomes/claves e valores irán sempre acoutados coa dobre comiña (excepto aqueles valores que sexan `true`, `false`, `null` ou *arrays* -como vimos no exemplo anterior-).
- Dado que os JSON son procesados coma *strings* en JS, é preciso acoutalos con comiña simple, para evitar o conflicto coa comiña dobre que empregamos no seu interior.
- Os valores non poden ser nin funcións, nin obxectos de tipo `date`, nin `undefined`.

Bibliografía

- Introducción a JSON: https://www.json.org/json-es.html

ÍNDICE

PROGRAMACIÓN ESTRUTURADA 3

HOISTING 3

MODO ESTRITO 4

FUNCIÓNS ANIÑADAS 5

CLOSURES 6

PROGRAMACIÓN ORIENTADA A OBXECTOS EN JS 9

OBXECTOS 9

CLASES 12

JSON 17

ESTRUTURA 17

MANIPULACIÓN 18

RESTRICIÓNS 18

BIBLIOGRAFÍA 19