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 serán o obxecto de estudio neste apartado.

Funcións

As funcións son bloques de código deseñados para levar a feito unha tarefa particular, de xeito que poden ser reempregados varias veces e executados fornecendo valores diferentes para obter distintos resultados.

En JS as funcións considéranse obxectos, polo que terán propiedades e métodos aos que podemos acceder para obter información:

```
function objectInfo() {
  return arguments.length;
}
typeof objectInfo; // function
objectInfo(3,4,5); // 3
objectInfo.toString(); // function objectInfo() { return
arguments.length; }
```

Unha función pode considerarse, segundo o seu ámbito de uso:

- **Método:** se é definida como unha propiedade dun obxecto.
- **Construtor:** se é definida para crear obxectos novos.

Funcións predefinidas

As funcións predefinidas son os métodos integrados no obxecto global da contorna de execución (p.ex.: nun navegador web o obxecto global será `window`). Estas son consideradas predefinidas posto que non é preciso facer referencia ao obxecto global para invocalas:

```
// Nun navegador web
alert("Hello!"); // equivalente a window.alert("Hello!");
```

Velaquí as funcións predefinidas de uso habitual nos navegadores web:

- **`decodeURI()`:** decodifica, coma o seu nome indica, un URI (Uniform Resource Identifier) fornecido coma *string* substituíndo as secuencias de escape UTF-8 (excepto as correspondentes con: `, / ? : @ & = + $ #`) propias dos navegadores web polos caracteres correspondentes.

```
decodeURI("search.php?place=Ca%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao
"); // search.php?place=Cañón del río Mao
```

- **`decodeURIComponent()`:** decodifica todo o ou parte dun URI fornecido coma *string* substituíndo as secuencias de escape UTF-8 (excepto as correspondentes con: `, / ? : @ & = + $ #`) propias dos navegadores web polos caracteres correspondentes..

```
decodeURIComponent("search.php%3Fplace%3DCa%C3%B1%C3%B3n%20del%20r
%C3%ADo%20Mao"); // search.php?place=Cañón del río Mao
```

- **`encodeURI()`:** codifica un URI fornecido coma *string* substituíndo os caracteres especiais (excepto: `, / ? : @ & = + $ #`) polas secuencias de escape UTF-8 correspondentes, para que sexa axeitado para os navegadores web.

```
encodeURI("search.php?place=Cañón del río Mao"); //
search.php?place=Ca%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao
```

- **`encodeURIComponent()`:** codifica todo ou parte dun URI fornecido coma *string* substituíndo os caracteres especiais (excepto: `, / ? : @ & = + $ #`) polas secuencias de escape UTF-8 correspondentes, para que sexa axeitado para os navegadores web.

```
encodeURIComponent("search.php?place=Cañón del río Mao"); //
search.php%3Fplace%3DCa%C3%B1%C3%B3n%20del%20r%C3%ADo%20Mao
```

- **`eval()`:** avalía e executa o código proporcionado como *string* no caso de que sexa código JS válido.

```
x = eval("((3+4)/2).toFixed(2)"); // 3.50
```

- **`isFinite()`:** determina se un valor é un número finito. Polo tanto, devolve `false` se o valor é `+Infinity`, `-Infinity` ou `NaN`. En calquera outro caso devolve `true`.

```
isFinite(1); // true
isFinite(1/0); // false
isFinite(+Infinity); // false
isFinite(-3.14); // true
isFinite(new Date()); // true
isFinite("2019-11-22"); // false
```

- **`isNaN()`:** determina se un valor non é un número legal devolvendo `true`/`false`.

```
isNaN(1); // false
isNaN(1/0); // false
isNaN(NaN); // true
isNaN(-3.14); // false
isNaN(new Date()); // false
isNaN("2019-11-22"); // true
```

- **`Number()`:** converte o valor dun obxecto a un número. Devolve `NaN` se non é válido.

```
Number(true); // 1
Number(new Date()); // 1576224556541
Number("3.14"); // 3.14
Number("3 14"); // NaN
```

- **`parseFloat()`:** determina se o primeiro caracter dunha *string* é un número (omitindo espacios en branco ao principio). Nese caso colle caracteres até que atope un non numérico e convérteo nun número decimal. Se non atopa ningún caracter numérico devolve `NaN`.

```
parseFloat("3") // 3
parseFloat("3.0") // 3
parseFloat("3.14") // 3.14
parseFloat("3 4 5") // 3
parseFloat("   3   ") // 3
parseFloat("70 kg") // 70
parseFloat("I am 70") // NaN
```

- **`parseInt()`:** determina se o primeiro caracter dunha *string* é un número (omitindo espacios en branco ao principio). Nese caso colle caracteres até que atope un non numérico e convérteo nun número enteiro. Se non atopa ningún caracter numérico devolve `NaN`.

  A función recibe un segundo parámetro opcional que indica a base (entre 2 e 36) na que está especificado o número.

```
parseInt("3") // 3
parseInt("1001", 2) // 9
parseInt("3.14") // 3.14
parseInt("3 4 5") // 3
parseInt("   3   ") // 3
parseInt("70 kg") // 70
parseInt("I am 70") // NaN
```

- **`String()`:** converte o valor dun obxecto nunha *string*. Equivale a empregar o método `toString()` nos obxectos en cuestión.

```
String(3+4); // 7
String(Boolean(1)); // "true"
String(new Date()); // Fri Dec 13 2019 09:28:51 GMT+0100
```

Funcións definidas polo usuario

Unha boa práctica de programación é a división do código en partes máis pequenas de xeito que, na medida do posible, sexan dedicadas a tarefas atómica, isto é, o máis concretas posible e que poidan ser empregadas sen coñecer como están deseñadas internamente e, desta forma, faciliten a súa reutilización e mantemento.

A programación estruturada fornece un mecanismo para acadar o anterior: as funcións definidas polo usuario. A súa declaración en JS ten as seguintes partes, por orde:

- **Palabra clave `function`.**
- **Nome da función (opcional):** seguindo as mesmas regras que o nomeado das variables: pode conter letras ASCII, díxitos, o guión baixo (_) e o dólar ($).
- **Parénteses (`()`):** para conter os parámetros de entrada.
- **Parámetros (opcionais) dentro dos parénteses:** compórtanse como variables locais á función. Os valores que se fornezan a estes parámetros no intre de chamar á función son coñecidos como argumentos.
- **Chaves (`{}`):** para delimitar o bloque de código da propia función.
- **Bloque de código (opcional):** contido dentro das chaves. Ao seu tempo pode incluír:
  - Instrución `return` (opcional): como mecanismo para devolver un valor. Supón a saída inmediata da función sen executar o posible código que veña a continuación.

```
function [faiAlgo]([parámetro1, parámetro2, ...]) {
  // Código a executar
  [return true;]
}
```

NOTA: os corchetes (`[]`) indican partes opcionais na declaración.

Funcións estándar

Aquelas que seguen a sintaxe tradicional e, polo tanto, teñen que levar nome.

```
function greet(a,b) {
  return a+" "+b+"!";
} // Función estándar
greet("Hello", "World"); // Hello World!
```

Funcións anónimas

Aquelas que non posúen nome propio. Velaquí os escenarios típicos nos que vamos atopalas:

- Para asignar a variables e poder invocalas posteriormente.

```
var minhaFuncion = function (a,b) { return a+", "+b+"!"; } //
Función anónima
minhaFuncion("Como vai", "amig@"); // Como vai, amig@!
```

- Pasadas como parámetro de métodos que reciban unha función, co gaio de realizar algún procesamento.

```
[21,3,5,13,8,44].filter(function (a) {return a>10;}); //
[21,13,44]
```

Construtor de funcións

No canto de empregar a palabra reservada `function`, pódese facer o mesmo co construtor do obxecto `Function()`.

```
var novaFuncion = new Function ("a","b","return a+', '+b+'!';");
// Función construtor
novaFuncion ("Chao", "xente"); // Chao, xente!
```

NOTA: este é un método moi útil para crear funcións alternativas dinamicamente en base ás respostas do usuario.

Funcións frecha

Consideradas simplemente como un xeito «reducido» de escribir funcións (o que coloquialmente no mundo da programación é coñecido coma «azucre sintáctico»), de xeito que posibilita empregar unha sintaxe máis curta para expresar as funcións. Mediante o mecanismo das funcións frecha elimínase a necesidade de empregar as palabras clave `function`, `return` e as chaves (`{}`):

```
var x = function (a,b) {return a+b}; // ECMAScript 5
x(3,4); // 7
const y = (a,b) => a+b; // ECMAScript 6
y(3,4); // 7
```

As súas características máis salientables son:

- Non teñen o seu propio obxecto `this`, polo que non son axeitadas para definir métodos de obxectos.
- Non admiten *hoisting*, polo que deben ser definidas sempre denantes ser empregadas.
- É recomendable empregar `const` no canto de `var` para declarar aquelas variables ás que se lles vaia asignar valor mediante funcións frecha, porque estas últimas sempre teñen valor constante.
- Só se pode omitir o uso de `return` e as chaves (`{}`) se a función é unha expresión simple, polo que é un bo hábito usalos sempre.

```
const z = (a,b) => {return a+b};
z(3,4); // 7
```

Invocación de funcións

Coñecese como **invocación** ao procedemento mediante o cal chamamos ás funcións para a súa execución. Para invocar unha función empregamos o operador parénteses (`()`). Faise así referencia ao valor devolto pola función xa que, de non engadir os parénteses, o que estariamos a facer sería referencia ao obxecto función:

```
function faiAlgo() {return "Ola mundo!";}
console.log(faiAlgo()); // Ola mundo!
console.log(faiAlgo); // function doSomething() {return "Ola
mundo!";}
```

A chamada dunha función pode ocorrer en distintas circunstancias:

- **Función global:** cando a función é chamada expresamente dende o propio código JS.

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

- **Evento web:** acontece cando un evento é desencadeado nunha páxina web (p.ex: o usuario preme un botón que ten asociada unha función).

```
<button id="action" onclick="faiAlgo();">Invocar!</button>
```

- **Autoinvocación:** prodúcese automaticamente no intre de definir a función. Para elo débese usar o operador parénteses (`()`).

```
(function faiAlgo() {return "Ola mundo!";})(); // Ola mundo!
(function () {return "Ola mundo!";})(); // Ola mundo! (función
anónima)
(function (a,b) {return a+b;})(3,4); // 7 (chamada con parámetros)
```

- **Método:** cando a función pertence a un obxecto haberá que antepoñer o nome do mesmo para invocala.

```
window.alert("Ola mundo!");
```

- **Construtor:** trátase dun tipo especial de función empregado para crear obxectos. Invócase mediante a palabra reservada `new`.

```
function meuArray(a,b) { return [a,b]; } // Función construtor
var x = new meuArray(3,4); // Invocación da función construtor
console.log(x.length); // 2
console.log(x); // [3,4]
```

Parámetros

Os parámetros son os mecanismos empregados polas funcións para recibir valores cos que traballar. Estes inclúense en forma de listaxe de valores separadospor comas na definición das funcións. Pola contra, os valores reais recibidos á hora de invocar a función coñécense como argumentos. Desta forma, os parámetros funcionan como variables locais á función, e son inicializados automaticamente cos valores pasados (argumentos) no intre da invocación.

Na definición das funcións non se especifica o tipo de datos dos parámetros e, polo tanto, o intérprete de JS non fai ningunha comprobación de:

- **Tipo dos argumentos:** será tarefa da programadora ou programador realizar as comprobacións e conversións de tipos que correspondan.
- **A cantidade dos argumentos.:**
    - Se unha función é chamada con menos argumentos dos declarados, asignaráselles un valores `undefined`. Para evitar isto pódense establecer valores por defecto para os parámetros na definición da función (posibilidade incorporada en ECMAScript 2015 ES6).

- Se unha función é chamada con máis argumentos dos declarados, simplemente descártanse os sobrantes.

```
function probarParametros1(a, b) { return b; }
probarParametros1(3); // undefined
function probarParametros2(a, b=1) { return b; }
probarParametros2(3); // 1
function probarParametros3(a, b) {return b;}
probarParametros3(3,4,5); // 4
```

As funcións JS teñen incorporado un obxecto `arguments` que contén un *array* cos argumentos recibidos na invocación da función.

```
function amosarArgumentos() {
  for (var i=0; i<arguments.length; i++) {
    console.log(arguments[i]);
  }
}
amosarArgumentos(3,4,5); // 3 4 5
```

En JS o paso de parámetros de tipo simple faise por valor, o que quere decir que, se dentro dunha función múdase o valor dun argumento, non é mudado o valor do parámetro orixinal (os cambios nos argumentos non son visibles fora das funcións).

```
var a=3,b=4;
console.log(faiAlgo(a,b)); // 8
console.log(a); // 3
console.log(b); // 4

function faiAlgo(a, b) {
    a += 1; // Modifica o argumento 'a' que funciona como variable
local
    return a+b;
}
```

Non obstante, se o que se pasa á función é un obxecto (como un *array*), o valor que se pasa é a referencia ao obxecto e, polo tanto, se dentro da función é mudada algunha propiedade do obxecto, si muda o valor orixinal do mesmo (os cambios nas propiedades de obxectos si son visibles fora das funcións).

```
// PASO POR VALOR
var num = 3; // O dato simple pasarase por valor
function incrementar(a) { return ++a; } // Modifica o argumento
'a' pero non a variable 'num' de fora
console.log(incrementar(num)); // 4
console.log(num); // 3
```

```
// PASO POR REFERENCIA
var numArray = [3,4,5]; // Obxecto array pasarase por referencia
function incrementarArray(a) {
    for (var i in a) { a[i]++; } // Modifica o argumento 'a' e a
variable 'numArray' de fora
    return a;
}
console.log(incrementarArray(numArray)); // [4,5,6]
console.log(numArray); // [4,5,6]
```

Hoisting

É un concepto 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
localScope();
console.log(i); // Error de referencia a variable global

function localScope() {
    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";
localScope();
console.log(i); // Punto non acadado

function localScope() {
    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
}
```

Closures

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

Para comprender mellor o concepto de *closure* supoñamos que queremos facer un contador de *clicks* 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 inicialízase 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 inicializar o contador que, ademais, pode considerarse unha variable privada ao ser accedida só 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.

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", age:24 }
persoa.faceAnos.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 *hosting* 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");
```

ÍNDICE