JavaScript (ii)

Miguel Ortuño
Escuela de Ingeniería de Fuenlabrada
Universidad Rey Juan Carlos

Marzo de 2024

Números aleatorios

```
'use strict'
// Math.floor() trunca un número real
console.log(Math.floor(9.99));  // 9

// Math.round () redondea un número
console.log(Math.round(9.5));  // 10
console.log(Math.round(-9.4));  // -9
console.log(Math.round(-9.5));  // -9
console.log(Math.round(-9.6));  // -10

// Math.random() devuelve un número real
// aleatorio entre 0 (inc) y 1 (excl)
console.log(Math.random());

let k = 10;
let x;

//Número entero aleatorio entre 0(inc) y k (excl)
x = Math.floor(k * Math.random());
console.log(x);
```

http://ortuno.es/ej_random.js

Fecha y Hora

La fecha (con su hora) se guarda en objetos de tipo `Date`

- Siempre es una hora UTC (Coordinated Universal Time), anteriormente llamada hora de Greenwich
- Se guarda como milisegundos transcurridos desde el *epoch*, 1 de enero de 1970
  - Es recomendable almacenar, procesar y transmitir este formato en todo momento, y solo inmediatamente antes de presentarlo al usuario, realizar la conversión necesaria
- Atención, no es el *tiempo unix* (segundos transcurridos desde el *epoch*), como en otros sistemas. Aquí son milisegundos
- Hay muchas páginas web que permiten conocer y manipular el tiempo unix. Busca en internet *unix time converter*

```
'use strict'

// Si construimos un objeto Date sin pasar argumentos,
// obtenemos la fecha actual
let d1 = new Date();
console.log(d1); // 2020-04-22T10:24:16.528Z

// Hacemos un cálculo matemático cualquiera, complejo,
// para tardar cierto tiempo (en js no hay nada equivalente
// a sleep)

for (let i = 0; i<100000000; ++i){
    let x = Math.random();
    x**2.50;
}

let d2 = new Date();
console.log(d2);  // 2020-04-22T10:24:17.555Z

// Tiempo transcurrido
console.log(d2-d1);  // 1027  (milisegundos)
```

Con frecuencia construiremos los objetos Date a partir de *intervalos*

- Un intervalo es un objeto de tipo *number* que contiene los milisegundos transcurridos desde el 1 de enero de 1970
- Conceptualmente es igual que un Date, pero es un tipo distinto (tipo *number*, no tipo *date*)
- Recuerda: muy parecido a una hora Unix, pero expresado en milisegundos

```
'use strict'
let intervalo = 0;
let d1 = new Date(intervalo);

console.log(intervalo, typeof(intervalo));
    // 0 'number'

console.log(d1, typeof(d1));
    // 1970-01-01T00:00:00.000Z 'object'
```

```
'use strict'

// Los intervalos son prácticos para hacer cálculos
let intervalo1 = 0;
let intervalo2 = intervalo1 + 24 * 60 * 60 * 1000;
     // ms en un día

console.log(intervalo1);   // 0
console.log(intervalo2);   // 86400000
let d1 = new Date(intervalo1);
console.log(d1);           // 1970-01-01T00:00:00.000Z
let d2 = new Date(intervalo2);
console.log(d2);          // 1970-01-02T00:00:00.000Z
```

Si necesitamos construir un *date* a partir del año, mes, día, etc, podemos pasar estos argumentos al constructor

- Pero cuidado, estos parámetros deberán ser hora local
- Si necesitamos una hora concreta a partir de la hora UTC, es necesario construir un intervalo con `Date.UTC()` y luego usarlo para construir *date*

```
// El constructor 'Date' devuelve un objeto Date, a partir de
// la hora local. P.e. 3 de septiembre de 2018, a las 9 de
// España. Cuidado: 0 es enero, 11 es diciembre
d = new Date(2018, 8, 3, 9, 0, 0);
console.log(d); // 2018-09-03T07:00:00.000Z   ZULU time (UTC)

// Para crear un Date desde hora UTC, usamos la función Date.UTC
// Atención: no devuelve un Date, devuelve un intervalo que
// debemos usar para crear el Date
let intervalo3 = Date.UTC(2018, 8, 3, 9, 0, 0);
console.log(intervalo3);  //  1535965200000

// Ahora construyo el objeto Date desde el intervalo
let d3 = new Date(intervalo3)
console.log(d3);  // 2018-09-03T09:00:00.000Z

// No se puede construir el objeto directamente
// d3 = new Date.UTC(2018, 8, 3, 9, 0, 0);
// TypeError: Date.UTC is not a constructor
```

```
d=new Date(2017, 8, 1, 9, 0, 0);  // 9:00 hora española
                                  // Mes 8: septiembre

// Acceso a cada unidad, expresada como hora local
console.log(d.getFullYear());   // 2017
console.log(d.getMonth());  // 8 (septiembre)
console.log(d.getDate());  // 1
console.log(d.getDay()); //5  (viernes)
console.log(d.getHours()); // 9  hora española
console.log(d.getMinutes()); // 0
console.log(d.getSeconds()); // 0
console.log(d.getMilliseconds()); //

// Acceso a cada unidad, expresada como hora UTC
console.log(d.getUTCFullYear());   // 2017
console.log(d.getUTCMonth());  // 8 (septiembre)
console.log(d.getUTCDate());  // 1
console.log(d.getUTCDay()); //5  (viernes)
console.log(d.getUTCHours()); // 7  hora UTC
console.log(d.getUTCMinutes()); // 0
console.log(d.getUTCSeconds()); // 0
console.log(d.getUTCMilliseconds()); // 0
```

Cálculo del tiempo transcurrido entre dos fechas

```
'use strict'
let d1,d2;

// 1 de septiembre de 2017, a las 9 de España
d1=new Date(2017, 8, 1, 9, 0, 0);

// las 9 y 10
d2=new Date(2017, 8, 1, 9, 10, 0);

console.log(d2-d1) // 600000 (600 segundos)
```

```
'use strict'
let d1,d2,s, ms_año, edad;

//  8 de julio de 1996, creación de la URJC
d1=new Date(1996, 6, 8, 0, 0, 0);
d2=new Date();
console.log(d1); //  1996-07-07T22:00:00.000Z
console.log(d2); // 2020-04-20T15:04:35.097Z

s= (d2-d1) ;  // milisegundos transcurridos

ms_año= 31536000000  // 1000*60*60*24*365
edad= (s/ms_año).toFixed(2);  // Redondeo a 2 decimales

console.log('Edad de la URJC:', edad);   // 23.80
```

Hora Unix a partir de Date

```
'use strict'

let d1 = new Date();
console.log(d1);   // 2020-04-22T13:38:34.777Z

// Basta con restar 0 y convertir a segundos
console.log( (d1-0) / 1000);       // 1587562714.777

// O usar el método getTime(), que es equivalente
console.log(d1.getTime() / 1000);  // 1587562714.777
```

Sabemos que restando dos Dates obtenemos un intervalo.

```
'use strict'
let d1 = new Date(2024, 2, 18, 10, 0, 0);
let d2 = new Date(2024, 2, 18, 10, 30, 0);
console.log(d1);  // 2024-03-18T09:00:00.000Z
console.log(d2);  // 2024-03-18T09:30:00.000Z
let intervalo = d2 - d1
console.log(typeof(intervalo), intervalo);
    // number 1800000  (180000 ms = 1800 s = 30 m)
```

Podríamos pensar que si a un Date le sumamos un número obtendríamos otro Date. P.e. a d2 le sumamos 30 minutos y obtenemos un date del mismo día, a las 11:00 (local).

```
let d3 = d2 + intervalo
console.log(d3)
// Mon Mar 18 2024 10:30:00 GMT+0100 (Central European Standard
↪  Time)1800000
```

¡Pero no!. Lo que hace es concatenar la cadena con la hora y la cadena con los milisegudos

Para sumar un intervalo a un date es necesario obtener la hora Unix (en ms), sumarle el intervalo y volver a convertir en un Date

```
'use strict'
let d2 = new Date(2024, 2, 18, 10, 30, 0);
console.log(d2); // 2024-03-18T09:30:00.000Z

let intervalo = 30 * 60 * 1000 // 30 minutos
console.log(intervalo) // 1800000

let d3 = new Date(d2.getTime() + intervalo)
console.log(d3)  // 2024-03-18T10:00:00.000Z
```

Módulos

Un módulo es

- Un fichero que contiene código JavaScript
- Que es invocado desde otro fichero distinto, que contiene el código principal

En otros lenguajes se les llama bibliotecas o librerías

En ECMAScript 5 y anteriores no había una forma nativa de usar módulos. Se desarrollaron diferentes herramientas, incompatibles entre sí, que permitían su uso
Las principales son:

- CommonJS
  Para Node.js
- RequireJS
  Para el navegador

En ECMASCript 6 sí hay soporte nativo para módulos, una mezcla de las dos sintaxis

- En la actualidad, año 2019, tanto en los navegadores como en node.js podemos usar módulos. Pero si el motor es un poco antiguo, esta funcionalidad de ECMAScript 6 puede no estar implementada

Extensiones de los ficheros

El mismo código ECMASCript 6 con módulos podemos ejecutarlo tanto en node.js como en el navegador. Con una diferencia:

- En el navegador, tanto los módulos como los ficheros que usan los módulos, tienen que tener extensión `.js`
- En node.js, tanto los módulos como los ficheros que usan los módulos, tienen que tener extensión `.mjs`

Una solución es usar los enlaces simbólicos de Unix (Linux, macOS). O los accesos directos de Windows

1. Creamos los ficheros con extensión `.mjs` para node.js
2. Para el navegador, creamos enlaces simbólicos `.js`

```
ln -s modulo.mjs modulo.js
ln -s programa.mjs programa.js
```

Si lo hacemos al revés (el fichero original con `.js` y enlace con `.mjs`), no funciona. Node.js toma el `js`, dando error

modulo.mjs

```
// Anteponemos 'export' a las funciones a exportar
export function f(){
    return "efe";
}

export function g(){
    return "ge";
}

// Esta función no la exportamos
function h(){
    return "hache";
}
```

programa.mjs

```
'use strict'
import * as modulo from './modulo';
console.log(modulo.f());
console.log(modulo.g());
```

Los módulos también se pueden usar de esta forma

```
'use strict'
import { f, g } from './modulo';
console.log(f());
console.log(g());
```

Pero aquí lo desaconsejamos, porque la llamada a una función no indica explícitamente de qué módulo viene

Observaciones

- El código dentro de un módulo siempre está en modo estricto, no es necesario indicarlo explícitamente
- Para especificar la ubicación del módulo, es imprescindible indicar el path relativo con `./`
- Si queremos que el código funcione tanto en el navegador como en node.js, no ponemos extensión

```
import * as modulo from './modulo';
import { f, g } from './modulo';
```

- Si sólo va a funcionar...
  - en node.js, podemos poner extensión `.mjs`

```
import * as modulo from './modulo.mjs';
import { f, g } from './modulo.mjs';
```

  - en el navegador, podemos poner extensión `.js`

```
import * as modulo from './modulo.js';
import { f, g } from './modulo.js';
```

Esto será necesario para probar nuestras prácticas de html canvas desde un servidor web local

Uso en node.js

- En las versiones actuales de node.js, el soporte para los módulos es experimental, hay que añadir un flag para interpretarlo

```
node --experimental-modules programa.mjs
```

Uso en el navegador

```html
<!DOCTYPE html>
<html lang="es-ES">

<head>
  <meta charset="utf-8">
  <title>Probando modulos</title>
</head>

<body>
  <script type="module" src="programa.js"> </script>
</body>

</html>
```

http://ortuno.es/probando_modulos.html

- En el elemento *script* añadimos el atributo *type=module*
  No porque `programa.js` sea un módulo, sino porque usa módulos

Un script que usa módulos, integrado en una página web solo puede ejecutarse cuando la página ha sido cargada desde un servidor web, no cuando ha sido leída de un fichero local

- Son restricciones de seguridad de CORS (*Cross-origin resource sharing*)

Para probar tus prácticas, tienes varias opciones
Opción 1: Web Server for Chrome

- Googlea *web server for Chrome*. Es una app para el navegador Chrome. Instálala
- Indícale el directorio a servir: el directorio donde están tus ficheros html
- Con esto tendrás disponibles tus ficheros en un servidor web local, accesible por omisión en `http://127.0.0.1:8887`

Opción 2: Servidor web local
P.e. usando python
`python -m SimpleHTTPServer <PUERTO> <DIRECTORIO>`
Los ficheros estarán accesibles en `http://localhost:<PUERTO>`

- En este caso, tendrás que importar módulos especificando la extensión `.js`, con lo que el código ya no funcionará en node.js
  Ejemplo:
  ```
  import * as vjcanvas from "./vjcanvas.js"
  ```
  en vez de
  ```
  import * as vjcanvas from "./vjcanvas"
  ```
  (Un servidor *de verdad* como p.e. apache o Nginx permite omitir la extensión. Pero el servidor web local necesita extensión explícita)
- `SimpleHTTPServer` es un módulo estándar de python, basta con tener python instalado

TypeScript

Como hemos visto, JavaScript tiene

- Características muy interesantes
- Muchos inconvenientes.
  Forman parte de la esencia del lenguaje, para evitarlos habría que usar un lenguaje diferente

Lenguajes alternativos a JavaScript hay muchos. Uno de los más convenientes es TypeScript

- TypeScript es un lenguaje *Open Source*, de alto nivel. Aparece en el año 2012, desarrollado y mantenido por Microsoft. Diseñado para hacer aplicaciones Web, en el cliente y en el servidor
- Es un superconjunto de JavaScript, al que añade tipado estático
  - En otras palabras: JavaScript es un subconjunto de TypeScript. Todo programa en JavaScript es también un programa en TypeScript

- El código fuente en TypeScript se *transpila* a JavaScript, de la misma forma que el código fuente de otros lenguajes se compila en código objeto / ejecutable. Por tanto, TypeScript funciona sobre cualquier navegador
- TypeScript permite detectar la práctica totalidad de problemas que *se le escapan* a JavaScript

POO basada en herencia vs POO basada en prototipos

La gran mayoría de lenguajes y herramientas diversas que emplean POO (programación orientada a objetos) aplican POO basada en herencia

- Es la forma tradicional. De hecho, es frecuente considerar que *POO* necesariamente implica herencia

En POO tradicional, hay un principio generalmente aceptado:
Favor object composition over class inheritance
(E. Gamma et al. *Design Patterns*. 1994)
La POO basada en prototipos va un paso más allá

- Para solucionar una serie de problemas bien conocidos en la POO tradicional. No *prefiere* la composición frente a la herencia, sino que omite por completo la herencia y se basa solo en composición.

Problemas de la herencia (1)

- Código yo-yo
  En jerarquías de cierta longitud, el programador se ve obligado a subir y bajar por las clases continuamente
- La herencia es una relación fuertemente acoplada. Los hijos heredan todo sobre sus padres, necesitan conocer todo sobre sus padres (y abuelos, bisabuelos...)
- Problema del gorila y el plátano
  Yo solo quería un plátano, pero me dieron el plátano, el gorila que sostenía el plátano y la jungla entera

Problemas de la herencia (2)

- La herencia es inflexible
  Por muy bien que se diseñe una jerarquía de clases, en dominios medianamente complejos acaban apareciendo casos no previstos que no encajan en la taxonomía inicial
- Herencia múltiple
  Heredar de diferentes clases es deseable, y teóricamente posible. Pero en la práctica resulta muy complicado
- Arquitectura frágil
  Rediseñar una clase obliga a modificar todos sus descendientes

POO basada en prototipos

- Aparece en el lenguaje Self, a mediados de los años 1980
- No hay clases como instancias de objetos: solo hay objetos, que se producen mediante funciones denominadas *factorías de objetos*
- Un objeto es una unidad de código que soluciona un problema concreto. No es necesario (o al menos no es crítico) pensar cómo usar variantes de ese objeto en otros casos
- A partir de ese objeto, según se va necesitando, se crean nuevos objetos añadiendo o eliminando las propiedades (datos y métodos) necesarias.
- Metáfora.
  Herencia: piezas de Ikea
  Prototipo: piezas de Lego

Problemas de la POO basada en prototipos (1)

- Poco conocida
  Pocos programadores la usan bien, pocos libros la explican bien
- No adecuada para principiantes
  Una vez que se conoce su uso resulta sencillo, pero su curva de aprendizaje es pronunciada
  Ejemplo: en JavaScript es necesario manejar al menos los siguiente conceptos:
  - Lambdas
  - Cierres
  - Funciones flecha
  - Prototipos
  - Factoria de objetos
  - Extensiones dinámicas de objetos: mezclas, composición, agregación

Problemas de la POO basada en prototipos (2)

- La flexibilidad del paradigma añade complejidad al intérprete/compilador, lo que afecta al rendimiento
  - Aunque los motores modernos son muy eficientes y este problema solo es relevante en casos extremos
- La flexibilidad del paradigma puede hacer más difícil el garantizar que los resultados sean correctos
  - Crítica análoga a la que hacen los partidarios de lenguajes de tipado estático frente a lenguajes de tipado dinámico
  - Una clase es un contrato estricto sobre lo que puede o no puede hacer un objeto. Estas restricciones no existen con los prototipos
    With great power comes great responsibility

Enlaces sobre POO basada en prototipos

- Composition over Inheritance
  Video en YouTube del canal *Fun Fun Funtions*

  `https://youtu.be/wfMtDGfHWpA`

- Master the JavaScript Interview: What's the Difference Between Class & Prototypal Inheritance?
  Eric Elliott

  `http://tinyurl.com/zbtjruf`

- Programming JavaScript Applications
  Eric Elliott. O'Reilly, 2015

  `http://proquest.safaribooksonline.com/book/programming/javascript/9781491950289`

- The Principles of Object-Oriented JavaScript
  Nicholas Zakas. No Starch Press, 2014

  `http://proquest.safaribooksonline.com/book/programming/javascript/9781457185304`

POO basada en herencia en JavaScript

El lenguaje JavaScript usa POO basada en prototipos.
Tras una cierta polémica, ECMASCript 6 soporta POO basada en herencia

- En realidad es *azúcar sintáctico*. Internamente siguen siendo prototipos

Los argumentos principales por los que se acepta este *paso atrás* son

1. Muchos programadores insisten en usarla. Distintas librerías, distintas soluciones ad-hoc. Es preferible una implementación oficial
2. Para principiantes con problemas sencillos, resulta más adecuado

Por este último motivo, en la presente asignatura veremos POO basada en herencia, no POO basada en prototipos

Clases

- Definimos clases con la palabra reservada `class`, el nombre de la clase y entre llaves, sus propiedades
- Por convenio, los nombres de clase empiezan por letra mayúscula
- A diferencia de lo que ocurre en los objetos, las propiedades no van separadas por comas
- Las propiedades de los objetos se crean en el método `constructor()`
- Se accede a las propiedes mediante la palabra reservada `this`, que representa al objeto
- Para crear una clase heredera de otra:

```
class Hija extends Madre{ ....}
```

- Para llamar a un método de la clase padre, se usa la palabra reservada `super`

```
'use strict'
class Circunferencia{
    constructor(x,y,r){
        this.x=x;
        this.y=y;
        this.r=r;
    }
    aCadena(){
        return '('+this.x+','+this.y+','+this.r+')';
    }
}
class Circulo extends Circunferencia{
    constructor(x,y,r,color){
        super(x,y,r);
        this.color=color;
    }
    aCadena(){
        return super.aCadena()+ " color:"+ this.color;
    }
}
let a=new Circunferencia(2,2,1);
console.log(a.aCadena());   // (2,2,1)
let b=new Circulo(2,2,1,"azul");
console.log(b.aCadena());  // (2,2,1) color: azul
```

Métodos

Para declarar los métodos de una clase no es necesario usar la palabra reservada `function`
Las clases tienen tres tipos de métodos.

- `constructor()`
  Es un método especial para crear inicializar los objetos de esta clase
- Métodos de prototipo (métodos *normales*)
- Métodos estático. (métodos *de clase*)
  Se crean anteponiendo la palabra reservada `static`
  Se invocan sin instanciar las clases en objetos, no pueden llamarse a través de una instancia de clase (a través de un objeto), solo a través de la clase
  ¿En qué casos un método debería ser estático?
  - Cuando tenga sentido sin que se haya declarado un objeto
  - Cuando procese 2 o más objetos, sin que uno tenga más relevancia que otro

```
'use strict'
class Circunferencia{
    constructor(x,y,r){
        this.x=x;
        this.y=x;
        this.r=r;
    }
    aCadena(){
        return '('+this.x+','+this.y+','+this.r+')';
    }
    static distanciaCentros( a, b){
        return Math.sqrt(
            Math.pow(a.x-b.x, 2) + Math.pow(a.y-b.y,
            2));
    }
    longitud(){
        return 2*Math.PI*this.r;
    }
}

let p=new Circunferencia(0,0,1);
let q=new Circunferencia(1,1,1);
console.log(p.aCadena());  // (0,0,1)
console.log( Circunferencia.distanciaCentros(p,q));
    // 1.4142135623730951
```

Enlaces sobre clases en JavaScript

- MDN Web Docs. Classes

  `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes`

- *Exploring ES6. Upgrade to the next version of JavaScript*
  Axel Rauschmayer

  `http://exploringjs.com/es6/ch_classes.html#sec_overview-classes`