UD6. PROGRAMACIÓN DE OBJETOS EN JAVASCRIPT

1. JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS

1.1. ¿QUÉ ES LA POO?

POO (Programación Orientada a Objetos) es un modelo de programación que cobró una enorme dimensión a finales de los 80. Cada objeto se programa de forma independiente, consiguiendo una modularización mayor que la propia de la programación modular. Los objetos son una estructura que aglutina datos y acciones (funciones).

La idea es que los objetos envían y reciben mensajes entre sí. Un objeto será un elemento en la aplicación capaz de recibir mensajes que provocan un procesamiento de dicho mensaje hasta producir un resultado que, a su vez, determinará un envío de mensajes a otro objeto.

El mantenimiento de las aplicaciones es más sencillo ya que nos concentramos en hacer que cada objeto funcione debidamente, y, cuando dicho objeto funciona, lo podemos utilizar en todas las aplicaciones que deseemos.

La idea de objeto es la de una estructura que aglutina datos (atributos o propiedades) y acciones (métodos). Si, por ejemplo, un objeto representara un coche, los datos podrían ser la marca, el color, la potencia, etc. Las acciones podrían ser arrancar, acelerar, repostar, ... es decir, lo que un coche es capaz de hacer.

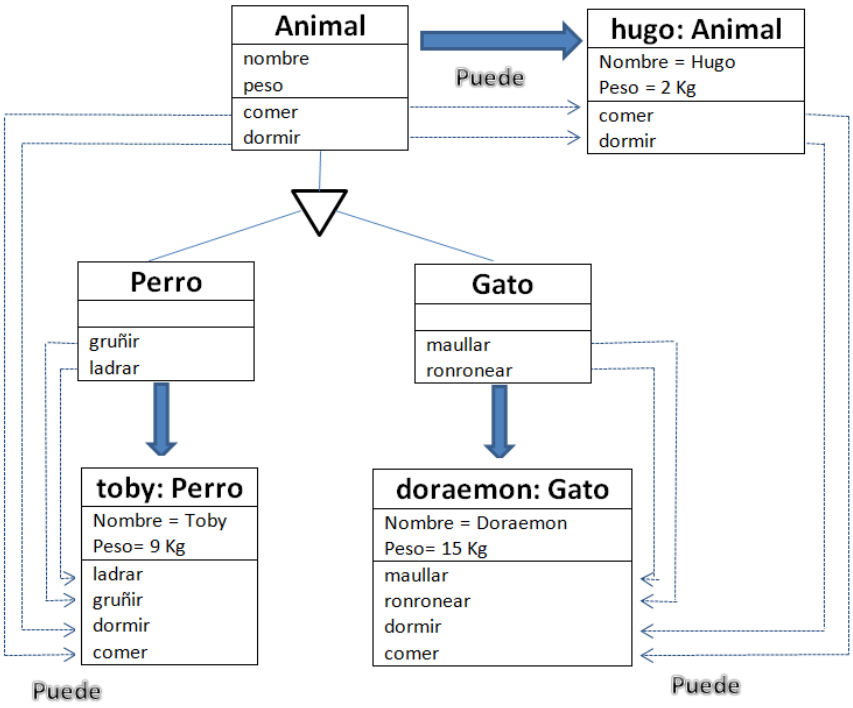
Una vez que un tipo de objeto (**clase**) se ha definido, se puede utilizar tantas veces como queramos y ya no nos preocuparemos por él. Programar aplicaciones consistirá en analizar y diseñar los objetos que necesitamos y cómo se han de comportar y comunicar entre sí.

1.2. CARACTERÍSTICAS DE LA POO

Hay cierto debate sobre si JavaScript un lenguaje orientado a objetos. Ese debate se debe a que se considera que otros lenguajes (como C++, Java o Python) implementan de forma más completa el paradigma de la POO.

La POO se dice que ha de tener estas características:

- **Abstracción**. Propiedad que permite reutilizar un objeto sin conocer su funcionamiento interno, nos bastará con que se nos proporcione lo que se conoce como su **interfaz externa**. La interfaz son las propiedades y métodos que permiten manipular y utilizar el objeto. En definitiva, la abstracción en la POO nos permite reutilizar el código de forma óptima, al poder utilizar objetos sin conocer los detalles de su implementación.
- **Encapsulamiento**. Capacidad de que los objetos puedan ocultar algunos métodos y propiedades de modo que, podamos elegir solo los que nos interesan que queden visibles.
- **Polimorfismo**. Esta característica permite conseguir que diferentes tipos de objetos puedan tener métodos con el mismo nombre, pero que actuarán de forma diferente. Por ejemplo, los objetos de clase círculo podrían tener un método llamado *suma* que permite unir su superficie a la de otro círculo, pero la clase **númeroGigante** podría tener el mismo método, pero para hacer sumas aritméticas.
- **Herencia**. Los diferentes tipos de objetos se suelen relacionar con **clases** de objetos. En la mayoría de lenguajes se definen clases y luego se crean objetos a partir de esas clases, es decir: objetos que son de una determinada clase. La herencia permite que haya clases que hereden las características de otras clases. Por ejemplo, la clase **Automóvil** puede terne un método que se llame **acelerar** y otro que se llame **frenar**. Si definimos una clase llamada **Coche**, indicando que esa clase deriva de la clase **Automóvil**, los coches también podrán frenar y acelerar.



Ejemplo de uso de clases y objetos con herencia en la POO

En la imagen, **Animal**, **Perro** y **Gato** son clases de objetos. Mientras que **toby**, **hugo** y **doraemon** son objetos ya concretos. **Perro** y **Gato** son clases que descienden de la clase **Animal**. El objeto **toby**, que es un perro, hace lo que hacen los perros (ladrar y gruñir), pero también heredan lo que hace cualquier animal (comer y dormir). Lo mismo ocurre con **doraemon**, hace cosas de gatos y también las cosas que comparte con cualquier otro animal. De **hugo** solo sabemos que es un animal genérico por ello solo puede hacer las cosas que hace cualquier otro animal.

1.3. JAVASCRIPT COMO LENGUAJE ORIENTADO A OBJETOS

Hay lenguajes que llevan a rajatabla las características vistas en los apartados anteriores, otros aportan cierta flexibilidad, o bien una reinterpretación de las mismas. Es todo un debate, a veces polémico, si JavaScript es un lenguaje realmente orientado a objetos o no.

Sin entrar a fondo en este debate, cierto es que JavaScript trabaja de forma diferente los objetos con respecto a lenguajes como Java o C++, pero indudablemente los objetos son importantísimos en JavaScript independientemente de cómo implementa el paradigma, toda aplicación JavaScript utiliza objetos y permite modular el código basándonos en interacciones entre objetos.

Sin duda, la gran diferencia de JavaScript está en el hecho de cómo maneja las clases y la herencia. Es posible programar objetos sin utilizar clases en JavaScript. En su lugar utiliza una orientación basada en prototipos. Ese mecanismo permite heredar características entre objetos gracias a considerar que cada objeto tiene un prototipo asociado que podemos modificar en cualquier momento para añadirle nuevas propiedades y métodos.

2. USO DE OBJETOS

2.1. ACCESO A PROPIEDADES Y MÉTODOS

Si tenemos un determinado objeto, el acceso a sus propiedades y métodos se consigue gracias al operador punto (.) Para acceder a una propiedad de un objeto, la sintaxis es:

```
objeto.propiedad
```

Cambiar el valor de una propiedad es posible, por ejemplo:

```
coche.color="Rojo";
```

Se ha modificado la propiedad color del coche. También podemos acceder usando esta forma:

```
objeto["propiedad"]
coche["color"]="Rojo";
```

Para utilizar métodos, la sintaxis es:

```
objeto.método([parámetros])
```

Ejemplo: coche.acelerar(25);

O bien, coche["acelerar"](25);

2.2. OBJETOS LITERALES

Los objetos más sencillos que podemos crear en JavaScript son los llamados literales o instancias directas. Son objetos en los que se pueden definir directamente sus propiedades y métodos sobre la marcha. Por ejemplo:

```
let punto=new Object();  // punto es un objeto, por ahora vacío
punto.x=5;  //Definimos la propiedad x y le damos valor
punto.y=punto.x*2;  //Definimos la propiedad "y" haciendo que valga el doble de lo que vale "x" :10
punto.mostrarCoordenadas=function(){
        return `(${punto.x},${punto.y})`;
}
console.log(punto.mostrarCoordenadas());
```

La primera línea del código anterior indica que la variable punto es un objeto. Una instrucción más sencilla equivalente a la anterior sería:

```
let punto={};
```

Las llaves permiten indicar que la variable punto es un objeto vacío. Tras esta instrucción podemos definir las propiedades y métodos de la misma forma. Otra posibilidad es crear el objeto y directamente asignarle propiedades y métodos:

```
let punto={
        x:19,
        y:36,
        mostrarCoordenadas : function(){
                return `(${punto.x},${punto.y})`;
}
```

```
};
console.log(punto.mostrarCoordenadas());
```

Esta forma de definir objetos se basa en la forma:

```
{
propiedad1 : valor1,
propiedad2 : valor2,
...
método1 : function(...){...},
...
}
```

El nombre de las propiedades y métodos puede ir entrecomillado, por lo que se admiten incluso nombres de propiedades y métodos con guiones y espacios en blanco (aunque no es muy recomendable hacerlo). El orden de las propiedades y métodos no es estricto, pero para una mejor legibilidad del código, es recomendable, definir primero las propiedades y luego los métodos.

```
let libro = {
        título :"Manual de UFOlogía",
        "n-serie": "187C2",
        autores: ["Pedro Martínes","Ana León"],  //Esta propiedad es un array
        editorial: {  //La editorial es otro objeto.
                nombre: "Inexistente S.A. ",
                País : "España"
        }
} ;
console.log(libro.título) ; // Escribe: Manual de UFOlogía
console.log(libro[n-serie]); //Escribe: 187C2
console.log(libro.editorial.nombre); // Escribe: Inexistente S.A.
```

Los objetos pueden ser todo lo complicados que queramos, una propiedad de un objeto puede ser un array, un mapa, un conjunto, otro objeto, etc.

2.3. OBJETO THIS

```
let punto={
        x:19,
        y:36,
        mostrarCoordenadas : function(){
                return `(${x},${y})`;
        }
};
console.log(punto.mostrarCoordenadas());
```

El resultado de estes código provoca un error de variable indefinida. La razón es que no hay una variable x, lo que hay es una propiedad **x** (lo mismo para la **y**). Por lo tanto, desde la función se asigna a **mostrarCoordenadas** no se puede acceder a ninguna variable llamada **x.**

```
let punto={
        x:19,
        y:36,
```

```
        mostrarCoordenadas : function(){
                return `(${punto.x},${punto.y})`;
        }
};
console.log(punto.mostrarCoordenadas());
```

Al hacer referencia a **punto.x** y **punto.y** ya no hay indefinición, está claro que ahora, sí sabemos que nos referimos a las propiedades del punto.

Pero veamos, finalmente este código:

```
let punto={
        x:19,
        y:36,
        mostrarCoordenadas : function(){
                return `(${this.x},${this.y})`;
        }
};
console.log(punto.mostrarCoordenadas());
```

Ahora, el resultado es el mismo. ¿A quién hacer referencia **this**?. Al objeto actual, en este caso es accesible a través de la variable **punto**, pero con código más complejo no sería tan fácil llegar al objeto. **This**, nos permite abstraernos y llegar al objeto propietario de la propiedad o el método que estamos definiendo, sea el que sea.

Veamos el siguiente código:

```
function doblarX(){
    this.x*=2;
}
let punto={
    x :15,
    y:7,
    doble:doblarX
};
let incógnita={
    x:5,
    dbl:doblarX
};
punto.doble();
licógnita.dbl();
console.log(punto.x); // Escribe 30
console.log(incógnita.x);  // Escribe 10
```

La función **doblarX** sirve para multiplicar por 2 la propiedad **x** del objeto que sea propietario de esta propiedad. Cuando esta función se asigna al método doble del objeto punto lo que ocurrirá, cuando se la invoque, es que se doblará el valor **x** de la propiedad del punto. Lo mismo ocurre cuando asignamos la función **doblarX** al método **dbl** del objeto incógnita. Lo interesante es que la palabra **this** permite llegar al objeto en cuestión y, por eso, podemos definir métodos, propiedades y funciones de forma más abstracta, y que funcionan incluso en objetos distintos.

2.4. RECORRER LAS PROPIEDADES DE UN OBJETOS

Para poder recorrer las propiedades de un objeto el bucle idóneo es **for..in**. Ejemplo

```
for(let prop in punto){
    console.log(`${prop} tiene el valor ${punto[prop]}`);
}
```

Suponiendo que el objeto punto es el mismo que en los objetos anteriores, se obtendrá este resultado:

```
x tiene el valor 19
y tiene el valor 36
mostrarCoordenadas tiene el valor function(){
                    return `(${this.x},${this.y})`;
                }
>
```

Si no queremos mostrar las funciones, el código podría ser:

```
for(let prop in punto){
    if(typeof punto[prop]!= "function") {
            console.log(`${prop} tiene el valor ${punto[prop]}`);
    }
}
```

2.5. BORRAR PROPIEDADES DE OBJETOS

El operador **delete** permite eliminar propiedades de los objetos. Se usa indicando, detrás de la palabra **delete**, la propiedad a borrar. Ejemplo:

```
let objeto={
    x:18,
    y:-10,
    z :-1
};
delete objeto.z;
console.log(objeto.x); // Escribe 18
console.log(objeto.z); //Escribe undefined
```

En el código anterior, la propiedad z no existe tras la instrucción **delete**.

3. USO AVANZADO DE OBJETOS

3.1. CREAR OBJETOS A TRAVÉS DE CONSTRUCTORES

Hay una forma de crear objetos que, sin duda, será más del agrado de los programadores acostumbrados a utilizar lenguajes orientados a clases. Se trata de utilizar la función constructora.

Los constructores son un elemento muy conocido por los programadores de LOO. En estos lenguajes, los métodos sirven para generar nuevos objetos de un tipo en concreto en base a datos que se envían a esa función constructora.

JavaScript utiliza esta idea, pero de forma un poco diferente. Realmente, lo que ocurre es que todas las funciones tienen capacidad de devolver un objeto y que además disponemos del método **this**, visto anteriormente, para hacer referencia al objeto actual al que pertenece la función. Ambos elementos combinados con un operador llamado **new** permite crear nuevos objetos utilizando funciones constructoras.

El operador **new** genera nuevos objetos a partir de esa función. A este proceso se le conoce como **instanciar** objetos, generar ejemplares concretos (**instances**).

```
//Función constructora
function Punto(coordX,coordY){
    this.x=coordX;
    this.y=coordY;
    this.mostrarCoordenadas=()=> `(${this.x},${this.y})`
}
let a=new Punto(10,20);
let b=new Punto(-3,6);
console.log(a.mostrarCoordenadas());
console.log(b.mostrarCoordenadas());
```

El resultado es:

```
(10,20)
(-3,6)
>
```

El hecho de usar como nombre de la función la palabra **Punto**, con mayúsculas en la primera letra, es una formalidad opcional que se utiliza por parte de los programadores para hacer notar que la función sirve para construir tipos de objetos. En realidad, el nombre de la función podemos escribirlo como queramos, pero siempre es aconsejable seguir estas normas para que el código sea más legible.

La función lo que hace es manipular (y a la vez definir) las propiedades del objeto que se devuelve. De eso se encargar la palabra **this**, que es la que hace posible llegar a las propiedades concretas del objeto que se está creando, sabiendo que los valores de esas propiedades podrán ser distintas en cada objeto. Se puede observar en el código anterior, que el constructor define propiedades (x e y) pero también métodos (***mostrarCoordenadas***) que serán propiedades y métodos que poseerán todos los objetos creados con este constructor.

Tras definir la función, la utilizamos para crear 2 objetos distintos (**a** y **b**) que usan esa función gracias al operador **new**, pero ambos del mismo tipo. Cuando usamos el método ***mostrarCoordenadas***, la información que devuelve es distinta porque las coordenadas x e y son diferentes.

Es decir, **Punto** es un tipo de objeto (evitamos decir clase porque en JavaScript es una palabra controvertida) mientras que **a** y **b** son objetos de este tipo.

Un detalle que hay que saber, aunque es de conocimiento muy profundo del lenguaje JavaScript es que realmente los constructores son funciones normales. Pasan a ser constructores cuando se usan con el operador **new**. Para que quede claro que es así, basta con ejecutar este código:

```
console.log(typeof Punto);
```

El resultado es la palabra **function**. Es más, es perfectamente válido este código:

```
let r=Punto(10,20);
```

Al no usar la palabra **new**, no hemos creado un objeto, por lo que la variable r es indefinida porque la función en sí no devuelve valor alguno (no hay **return**). Dicho de otro modo, no tiene sentido usar una función constructora de esa forma.

3.2. OPERADOR INSTANCEOF

Existe un operador similar a **typeof**, pero pensado para ser usado con objetos, que se llama **instanceof**. La razón para usar este nuevo operador es que todos los objetos pertenecen al mismo tipo de datos: **Object**. A veces, necesitamos comprobar a qué tipo de objeto pertenece uno dado. Teniendo vigente el código del apartado anterior, si ejecutamos después este código:

```
console.log(b instanceof Punto); //Devuelve true, b es un Punto
console.log(b instanceof Object); //Devuelve true, b es un Objeto
```

La variable b hace referencia a un objeto creado con la función constructora **Punto**. Es decir, **b** es un punto. Pero todos los objetos, a su vez, pertenecen a un tipo básico llamado **Object**. Gracias a ello, sabemos que los objetos de tipo Punto heredan todo lo que posee la clase **Object**.

Hay otra posibilidad para obtener o comprobar el tipo de un objeto. Es una propiedad que se añadió a todos los objetos en la versión estándar ES2015, es **constructor.name**. La propiedad constructor la tienen todos los objetos y permite obtener información sobre la construcción del objeto. En realidad, *constructor* es también un objeto que entre sus propiedades posee la propiedad *name* que nos devolverá el nombre de la función constructora y, por lo tanto, el tipo de objeto. Ejemplo:

```
console.log(b.constructor.name); //Escribe Punto
```

3.3. PROTOTIPOS

3.3.1. IDEA DE PROTOTIPOS

Desde hace tiempo JavaScript utiliza un concepto muy interesante para conseguir implementar lo que en otros lenguajes se conoce como herencia.

En el lenguaje Java (no JavaScript) y en otros lenguajes, todo objeto pertenece a una clase. Para definir un objeto es obligatorio primero crear una clase. Además, se puede indicar que una clase es **heredera** de otra clase, por lo que esa clase dispondrá de las propiedades y métodos definidos en la clase de la que hereda.

JavaScript no nació con esa idea. La idea es que todos los objetos procedentes del mismo tipo de función constructora tienen un mismo prototipo con el que enlazan. El prototipo de un objeto es una serie de métodos y propiedades comunes.

En los lenguajes que usan clases, el código de los métodos se copia a los objetos de esa clase. Sin embargo, en JavaScript lo que se hace es enlazar con su prototipo. El prototipo es la parte común de los objetos del mismo tipo. Lo interesante en JavaScript es que podemos modificar el prototipo sobre la marcha, y los objetos que enlazan con ese prototipo inmediatamente estarán al día porque el enlace con su prototipo es dinámico.

En el código anterior, la variable **a** es un objeto de la clase **Punto**, al igual que la variable **b**. El acceso al prototipo de un objeto se puede hacer con la propiedad **__proto__** (hay 2 guiones al principio y al final de la palabra proto.). Si mostramos esa propiedad para la variable a:

```
console.log(a.__proto__);
```
Se nos muestra:

```
{}
```
Con ellos se nos dice que el prototipo de **a** no tiene definido ningún método ni propiedad. Los objetos de tipo punto toman las propiedades y métodos definidos en la función constructora, pero no hay propiedades comunes.

Una forma equivalente de obtener el prototipo es mediante el método **getPrototypeOf** que es un método de la clase genérico **Object**. Se usa de esta forma:

```
Object.getPrototypeOf(a)
```
El resultado será el mismo

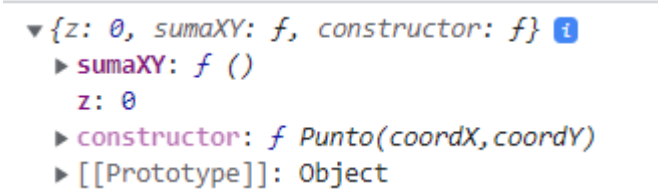
3.3.2. MODIFICADOR DE PROTOTIPOS

Para modificar prototipos basta con indicar la propiedad **prototype** y después definir propiedades y métodos a voluntad. Así si escribimos el código:

```
console.log(Punto.prototype);
```
Obtendremos el prototipo de la función Punto, que será un objeto vacío porque no hemos definido nada para él.

Para definir, por ejemplo, un nuevo método y una nueva propiedad podemos indicarlos y darles valor. Por ejemplo:

```
Punto.prototype.sumaXY=function(){
    return this.x+this.y;
}
Punto.prototype.z=0;
```
Hemos definido un método llamado **sumaXY** y lo hemos definido para el prototipo de los objetos basados en **Punto**. También hemos creado en ese mismo prototipo una propiedad llamada z con valor de cero. Si mostramos ahora el prototipo veremos:



Dará igual que lo hagamos con la expresión **Punto.prototype** o con (si **p** es un objeto tipo **Punto**) **p.__proto__**;

Ahora el prototipo de Punto ya tiene un método y una propiedad. Lo interesante es que todos los objetos basados en Punto disponen de esa propiedad y método:

```
let a=new Punto(10,20);
let b=new Punto(-3,6);
console.log(a.sumaXY()); //Muestra 30, resultado de sumar 10 +20
console.log(b.sumaXY()); //Muestra 3, resultado de suma –3+6
console.log(a.z); //Muestra 0
console.log(b.z);  //Muestra 0
```

Un detalle muy importante es lo que ocurre si modificamos en un objeto la propiedad heredada. Por ejemplo:

```
a.z=7;
```

Ahora la variable **a**, ya no coge la propiedad **z** del prototipo, tiene un valor propio de esa propiedad. Aunque modifiquemos la propiedad **z** a través del prototipo, la variable **a** no lo reflejará porque su propiedad **z** ya es independiente de su prototipo. Sin embargo, todos los demás objetos usarán la propiedad z del prototipo. Veamos el siguiente código:

```
console.log(a);
console.log(b);
```

El resultado sería:

```
▶ Punto {x: 10, y: 20, z: 7, mostrarCoordenadas: ƒ}
▶ Punto {x: -3, y: 6, mostrarCoordenadas: ƒ}
```

La primera línea del resultado muestra las propiedades y métodos de **a**, la segunda los de **b**. El objeto **a** ha definido una propiedad que no tiene **b**. No obstante, **b** la tiene.

```
console.log(b.z); //Muestra 0
```

La propiedad **z** en el caso de **b** la obtiene del prototipo. Lo mismo pasaría con los métodos. Si un objeto redefine un método, entonces, usa su versión de forma prioritaria sobre la del prototipo.

Pero siempre podemos acceder a las propiedades y métodos de los prototipos de un objeto:

```
Console.log(a.__proto__.z); //Escribe 0
```

Un detalle interesante es que podemos modificar el prototipo incluso de los objetos estándar. Ejemplo:

```
Array.prototype.obtenerPares=function(){
        return this.filter((x)=>(x%2==0));
}
let a=[1,2,3,4,5,6,7,8,9];
console.log(a.obtenerPares());
```

En el código anterior conseguimos que todos los arrays dispongan de un nuevo método llamado **obtenerPares**, el cual devuelve un nuevo array en el que solo quedan los números pares del array original.

3.4. NOTACIÓN JSON

JSON es el acrónimo de JavaScript Objects Notation que se creó en 2001 por parte de Douglas Crokford con la idea de aprovechar la forma que posee JavaScript para crear objetos estáticos para crear un formato documental que sea más cómodo y eficiente para los programadores que los lenguajes de marcado como XML.

Aunque inicialmente se creó para ser usado desde JavaScript, actualmente se considera un formato documental independiente de cualquier lenguaje. La información almacenada en formato JSON se puede manipular desde casi cualquier lenguaje de programación actual.

El formato JSON se basa en indicar propiedades y valores separados por (:) Los valores se indican igual que en JavaScript (textos entrecomillados, número tal cual y usando el punto decimal, etc.) Se permite el uso de arrays, anidar objetos, etc. Pero hay diferencias que reseñar:

JSON solo admite definir propiedades, no métodos

En JSON el nombre de las propiedades tiene que ir entre comillas dobles.

Ejemplo:

```
{
        "título" : "Manual de OFOlogía",
        "n-serie": "187C2",
        "autores" : ["Pedro Martinez", "Ana León"],
        "editorial" : {
                "nombre": "Inexistente S.A.",
                "país" : "España"
}
"edición": 2,
"ensayo": true
}
```

Manipular datos en JSON es muy importante en la creación actual de aplicaciones web, por ello, JavaScript aporta un objeto llamado JSON que permite manipular los datos en ese formato.

Este objeto posee 2 métodos:

- **stringify**. Sirve para convertir un objeto JavaScript a un string que contiene el formato JSON equivalente.
- **parse**. Recibe un texto en formato JSON y evalúa su corrección. Si es correcto, retorna el objeto equivalente y si no, devuelve una excepción de tipo **SyntaxError**.

Ejemplo de stringify:

```
const musico1={
```

```
        nombre: "Ed",
        apellido: "Sheeran",
        fecha_nacimiento: {
                día:17,
                mes:2,
                año: 1991
        } ,
        discos: ['+','x','÷','=']
}
console.log(JSON.stringify(musico1);
```

```
{"nombre":"Ed","apellido":"Sheeran","fecha_nacimiento":{"día":17,"mes":2,"año":1991},"discos":["+","x","÷","="]}
```

4. OBJETOS PREDEFINIDOS

El lenguaje JavaScript proporciona objetos ya preparados para ser utilizados en nuestro código. En realidad, ya conocemos bastantes: **Array**, **Map** o **Set** son algunos de ellos. Veremos otros que también nos serán útiles.

4.1. OBJETO MATH

Se trata de un objeto global que facilita la ejecución de algunas operaciones matemáticas de alto nivel.

4.1.1. CONSTANTES DE MATH

Permiten usar en el código valores de constantes matemáticas, por ejemplo, tenemos a Math.PI, que representa el valor matemático de PI. Podemos usarla en nuestro código de esta forma. Ejemplo:

```
function Circulo(r){
this.radio=r;
        this.calcularCircunferencia= ()=>(2*Math.PI*this.radio);
}
let c=new Circulo(10);
console.log(c.calcularCircunferencia()); //Escribe 62.83185307179586
```

En la siguiente tabla se muestra la lista completa de constantes:

CONSTANTE	SIGNIFICADO
E	Valor matemático de e
LN10	Logaritmo neperiano de 10
LN2	Logaritmo neperiano de 2
LOG10E	Logaritmo decimal de e
LOG2E	Logaritmo binario de e
PI	Constante PI

SQRT1_2	Resultado de la división de 1 entre la raíz cuadrada de 2
SQRT2	Raíz cuadrada de 2

4.1.2. MÉTODOS DE MATH

MÉTODO	SIGNIFICADO
abs(n)	Calcula el valor absoluto del número n
acos(n)	Calcula el arco coseno del número n
asin(n)	Calcula el arco seno absoluto del número n
atan(n)	Calcula el arco tangente del número n
ceil(n)	Redondea el número n (si es decimal) a su valor superior.
cos(n)	Coseno de n.
exp(n)	Número e elevado a n
floor(n)	Redondea el número n (si es decimal) a su valor inferior.
log(n)	Calcula el logaritmo decimal de n
max(a,b)	Siendo a y b 2 números, devuelve el mayor de ellos
min(a,b)	Siendo a y b 2 números, devuelve el menor de ellos
pow(a,b)	Potencia. Devuelve el resultado de a elevado a b
random()	Devuelve un número aleatorio entre 0 y 1
round(n)	Redondea n a su entero más próximo.
sin(n)	Devuelve el seno de n
sqrt(n)	Raíz cuadrada de n
tan(n)	Tangente de n

4.2. OBJETO DATE

Es el nombre de un tipo de objetos preparados para manejar fechas. Crear un objeto de fecha es usar el constructor de objetos Date. En la construcción más simple, la función Date no requiere parámetros:

```
let hoy=new Date();
console.log(hoy);
```

Escribiría en la consola algo como:

```
Wed Aug 17 2022 18:57:39 GMT+0200 (hora de verano de Europa central)
```

Un objeto de tipo **Date** representa un momento concreto de tiempo. El texto que aparece por consola indica que la variable **hoy** hace referencia a un objeto de tipo **Date** que almacena como fecha el 17 de agosto de 2020 (miércoles) a las 18:57 y, además, se indica que esa fecha está 2 horas por encima de la del meridiano de Greenwich y que el horario de esa zona en este momento es el que se usa en verano.

Los objetos Date se pueden crear indicando una fecha concreta de esta manera:

```
new Date=(año,mes,día,hora,minutos,segundos,ms);
```
Ejemplo:

```
let fecha=new Date(2004,6,29,0,40,0,0);
console.log(fecha);
```
Muestra:

```
Thu Jul 29 2004 00:40:00 GMT+0200 (hora de verano de Europa central)
```

Observando el resultado queda patente que el mes 6 es julio (en lugar de junio), porque se considera que el mes 0 es enero.

No es obligatorio indicar todos los datos, podemos indicar solo año, mes y día.

```
let fecha=new Date(2004,6,29);
```
Hay una tercera opción que es crear una fecha a partir de un número que simboliza el número de milisegundos transcurridos desde el 1 de enero de 1970. Ejemplo:

```
let fecha2=new Date(1000);
console.log(fecha2);
```
Saldría:

```
Thu Jan 01 1970 01:00:01 GMT+0100 (hora estándar de Europa central)
```

Ese resultado muestra un segundo tras el 1 de enero de 1970. Con el horario de esa fecha en la franja horaria de España, sería la 1 de la madrugada.

4.2.1. MÉTODOS DE LOS OBJETOS DATE

Los objetos de tipo Date disponen de numerosos métodos para realizar operaciones sobre fechas. Por ejemplo:

```
let fecha=new Date();
console.log(fecha.getFullYear()); //Escribe el año actual (p.ej: 2022)
```

MÉTODO	QUÉ HACE

getDate()	Devuelve el día del mes. Número entre 1 y 31
GetUTCDate()	Idem. La diferencia es que UTC usa la fecha global (lo que corresponda al meridiano de Greenwich)
getDay()	Devuelve el día de la semana. Entre 0 (domingo) y 6 (sábado)
getUTCDay()	Idem en formato universal
getFullYear()	Devuelve el año con 4 dígitos
getUTCFullYear()	Idem en formato universal
getHours()	Obtiene la hora de la fecha (número que va de 0 a 23)
getUTCHours()	Obtiene la hora en formato universal
getMilliseconds()	Devuelve los milisegundos entre 0 y 9999
getUTCMilliseconds()	Idem en formato universal
getMinutes()	Devuelve los minutos. Entre 0 y 59
getUTCMinutes()	Obtiene los minutos en formato universal
getMonth()	Devuelve el mes. Entre 0 (enero) y 11 (diciembre)
getUTCMonth()	Devuelve el mes en formato universal
getSeconds()	Devuelve los segundos. Entre 0 y 59
getUTCSeconds()	Devuelve los segundos en formato universal.
getTime()	Devuelve los milisegundos transcurridos entre el día 1 de enero de 1970 y la fecha correspondiente al objeto al que se le pasa el mensaje
getTimezoneOffset()	Obtiene el valor en milisegundos usando el formato universal.
setDate(día)	Actualiza el día del mes de la fecha. Usa el indicado en el parámetro
setUTCDate(día)	Versión en formato universal del método anterior
setFullYear(año)	Cambia el año de la fecha al número que recibe por parámetro
setUTCFullYear(año)	Versión en formato universal del método anterior
setHours(hora)	Actualiza la hora
setUTCHours(hora)	Versión en formato universal del método anterior
setMilliseconds(ms)	Establece el valor de los milisegundos

setUTCMilliseconds(ms)	Versión en formato universal del método anterior
setMinutes(minutos)	Cambia los minutos
setUTCMinutes(minutos)	Versión en formato universal del método anterior
setMonth(mes)	Cambia el mes (atención al mes que empieza por 0)
setUTCMonth(mes)	Versión en formato universal del método anterior
setSeconds(segundos)	Cambia los segundos
setTime(segundos)	Actualiza la fecha completa. Recibe un número de milisegundos desde el 1 de enero de 1970
toTimeString()	Convierte la parte de tiempo de un objeto Date en una cadena
toDateString()	Muestra la fecha en un formato más humano de lectura
toGMTString()	Igual que la anterior, pero antes de mostrarla convierte la fecha a la correspondiente según el meridiano de Greenwich.

toISOString()	Muestra la fecha en formato ISO el cual cumple esta plantilla: **yyyy-mm-ddThh:mm:ss:sssZ**
	Ejemplo:
	`var d=new Date();`
	`console.log(d.toISOString());`
	Obtiene por ejemplo (si son las 21:25:35,857 segundos del día 17 de agosto de 2022 en el meridiano de Greenwich)
	`2022-08-17T21:25:35.857Z`

toLocaleString([codigoLocal[,opciones]])	Sin parámetros muestra la fecha y hora en formato de texto usando la configuración local.
	`console.log(fecha.toLocaleString());`
	Obtiene, por ejemplo:
	`17/8/2022, 23:34:00`
	Pero es posible indicar el código del país.
	`console.log(fecha.toLocaleString("en"));`
	Ahora muestra:
	`8/17/2022, 11:34:00 PM`
	Las opciones permiten indicar formato numérico, zona horaria, formato de hora, etc. Ejemplo:
	`console.log(d.toLocaleString("es",{"timeZone":"UTC"}));`

toLocaleDateString()	Muestra la fecha (sin la hora) en formato de texto usando la configuración local.
toTimeString()	Muestra la hora (sin la fecha) en formato de texto usando la configuración local.

toString()	Muestra la fecha en formato de texto usando la configuración habitual de JavaScript.
toUTCString()	Versión en formato universal de la anterior
toJSON()	Muestra la fecha en formato JSON. Obtiene lo mismo que la anterior.

4.2.2. MÉTODOS ESTÁTICOS DE DATE

Estos métodos usan directamente el objeto global Date. Se usan en esta forma: Date.nombreMétodo.

La lista de métodos es:

MÉTODO	SIGNIFICADO
now()	Fecha actual en milisegundos desde el día 1 de enero de 1970
parse(objetoFecha)	Obtiene una representación en forma de texto de la fecha.
UTC(año,mes,día,horas,minutos ,segundos, ms)	Consigue, de la fecha indicada, la forma equivalente en milisegundos. Esta forma mostrará los milisegundos transcurridos desde el 1 de enero de 1970

4.3. EXPRESIONES REGULARES

4.3.1. ¿QUÉ SON LAS EXPRESIONES REGULARES?

La mayoría de los lenguajes de programación disponen de, al menos, alguna capacidad para trabajar con expresiones regulares. La cuestión es ¿para qué sirven estas expresiones?

Se utilizan, sobre todo, para establecer un patrón que permita establecer condiciones avanzadas en los textos, de modo que podamos validar los textos que encajen con ese patrón. Las labores típicas e las que ayudan las expresiones son: validación de errores, búsqueda de textos con reglas complejas y modificación avanzada de textos.

Por ejemplo, en España un número de identificación personal que se suele exigir a menudo es el NIF, formado por 8 números y una letra, cuando coincide con el DNI de las personas. Pero puede haber una letra en la primera cifra en otras circunstancias y esa letra solo puede ser K,L,X,Y o Z.

Validar todas esas posibilidades, aun sin contar que la letra final debe cumplir una condición matemática, es muy complejo. Pero una sola expresión regular puede establecer el patrón que cumple el NIF y así facilitar mucho su validación.

4.3.2. EXPRESIONES REGULARES EN JAVASCRIPT

Las expresiones regulares de JavaScript en realidad son objetos de tipo RegExp. Debido a la importancia que tienen estas expresiones, JavaScript ha desarrollado una sintaxis que facilita mucho su uso.

Las expresiones regulares se pueden crear indicando las mismas entre 2 barras de dividir. Tras las barras se pueden indicar modificadores, la sintaxis sería esta:

```
/expresionRegular/
```

Así, para crear un objeto que contenga una expresión regular podemos hacer lo siguiente:

```
let expNIF=/[KLXYZ0-9][0-9]{7}[A-Z] /;
```

En esa expresión no hemos utilizado ningún modificador, no es obligatorio hacerlo. Esa expresión es la que permite establecer el patrón general que cumplen los NIF.

Otra forma de crear la expresión, usando una notación más formal es:

```
let expNIF=new RegExp('[KLXYZ0-9][0-9]{7}[A-Z]') ;
```

Es más habitual usar la primera forma, por ser más rápida e incluso entendible.

4.3.3. ELEMENTOS DE LAS EXPRESIONES REGULARES

El patrón de una expresión regular puede contener diversos símbolos que se interpretan de forma especial. Para entender las posibilidades de estos símbolos debemos empezar a hablar del método **test** de las expresiones regulares. Este método devuelve verdadero si el texto que evaluamos con la expresión regular cumple el patrón.

Vamos a ver un ejemplo para intentar aclarar la idea:

```
let expresion=/c/;
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('Casa')); //Escribe false
console.log(expresion.test('barbacoa')); //Escribe true
```

La expresión **/c/** la cumple cualquier texto que tenga dentro una letra **c**. Por defecto, se distingue entre mayúsculas y minúsculas, por eso la palabra **Casa** no cumple la expresión.

Si queremos no distinguir entre mayúsculas y minúsculas, podemos añadir el modificador i al final de la expresión.

```
let expresion=/c/i;
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('Casa')); //Escribe true
console.log(expresion.test('barbacoa')); //Escribe true
```

Si en lugar de un carácter usamos varios, solo los textos que contengan esos caracteres en ese orden cumplirán el patrón.

```
let expresion=/as/;
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('Casa')); //Escribe true
console.log(expresion.test('asador')); //Escribe true
console.log(expresion.test('valiosa')); //Escribe false
```

La potencia de las expresiones regulares se dispara con el uso de símbolos especiales. Vemos a continuación los principales:

Que comience por

El símbolo circunflejo (^) obliga a que el siguiente símbolo de la expresión sea el primero que obligatoriamente, tenga el texto que validemos:

```
let expresion=/^c/;
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('barbacoa')); //Escribe false
```

Que termine por

El símbolo de dólar ($) hace que el carácter anterior sea, obligatoriamente, el último del texto.

```
let expresion=/a$/;
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('barbacoa')); //Escribe true
console.log(expresion.test('rio')); //Escribe false
```

Comodín de cualquier carácter

El punto (.) es un símbolo especial de las expresiones regulares que representan un carácter cualquiera. Ejemplo:

```
let expresion=/a.e/;
console.log(expresion.test('caserio')); //Escribe true
console.log(expresion.test('aereo')); //Escribe false
console.log(expresion.test('alambique')); //Escribe false
```

En el código anterior, la expresión /a.e/ la cumplen los textos con una letra a seguida de otro carácter (el que sea, pero uno) y luego la letra e.

Símbolos opcionales

Cuando se pone una serie de símbolos entre corchetes, se permite cualquier de ellos. Por ejemplo, la expresión **/[arft]/** la cumpliría cualquier texto con una letra **a** o una **r** o una **f** o una **t**. Gracias a ello podemos conseguir códigos como este:

```
let expresion=/[ao]$/; // Es válido cualquier texto que acabe por "a" u "o"
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('barbacoa')); //Escribe true
console.log(expresion.test('rio')); //Escribe true
```

Podemos incluso indicar rangos. Por ejemplo, la expresión **[a-z]** significa cualquier carácter entre la letra **a** y la **z** minúscula.

```
let expresion=/^[a-z]/; // Es válido cualquier texto que empiece con minúscula
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('Casa')); //Escribe false
console.log(expresion.test('rio')); //Escribe true
console.log(expresion.test('árbol')); //Escribe false
```

El carácter "á" en Unicode no está entre la a y la z.

```
let expresion=/^[a-záéíóú]/; // Es válido cualquier texto que empiece con minúscula
console.log(expresion.test('casa')); //Escribe true
console.log(expresion.test('Casa')); //Escribe false
console.log(expresion.test('rio')); //Escribe true
console.log(expresion.test('árbol')); //Escribe true
```

Carácter no permitido

Si dentro del corchete indicamos un carácter circunflejo, estamos indicando que el contenido no se tiene que cumplir. Por ejemplo **[^ae]** significa que no puede aparecer ni la letra **a** ni la **e**, cualquier otra será válida. Ejemplo:

```
let expresion=/^[^AEIOU]/; // Es válido cualquier texto que empiece con minúscula
console.log(expresion.test('Empresa')); //Escribe false
console.log(expresion.test('Casa')); //Escribe true
console.log(expresion.test('Ala')); //Escribe false
```

La expresión **/^[^AEIOU]/** significa que **no** empiece por ninguna vocal en mayúsculas.

Símbolos de cardinalidad

Son símbolos que sirven para indicar la repetición de una expresión. Son, concretamente:

SÍMBOLO	SIGNIFICADO
x+	La expresión a la izquierda de este símbolo se puede repetir 1 o más veces
x*	La expresión a la izquierda de este símbolo se puede repetir 0 o más veces
x?	La expresión a la izquierda de este símbolo se puede repetir 0 o 1 veces
x{n}	La expresión x aparecerá n veces, siendo n un número entero positivo
x{n,}	La expresión x aparecerá n o más veces
x{m,n}	La expresión x aparecerá de m a n veces

Ejemplos:

```
let expresion=/^[AEIOU].+a/;
console.log(expresion.test('Asa')); //Escribe true
console.log(expresion.test('Estación')); //Escribe true
console.log(expresion.test('Ea')); //Escribe false
```

La última expresión es falsa porque debería haber un símbolo entre la E y la a. En cambio:

```
let expresion=/^[AEIOU].*a/;
console.log(expresion.test('Asa')); //Escribe true
console.log(expresion.test('Estación')); //Escribe true
console.log(expresion.test('Ea')); //Escribe true
```

Si quisiéramos validar un código postal español, debemos indicar que solo se permiten 5 caracteres numéricos.

```
let CPválido = /^[0-9]{5}$/;
```

Paréntesis

Los paréntesis permiten agrupar expresiones. Eso permite realizar expresiones aún más complejas. Por ejemplo:

```
let expresion = /([a-z]{2}[0-9]){3}/;
```

Esto obliga a que el texto tenga 2 letras de la a **a** la **z**, luego un número. El último número entre llaves hace referencia a todo el paréntesis, por lo que las 2 letras y el número deberán aparecer 3 veces seguidas. Uso posible:

```
let expresion= /([a-z]{2}[0-9]){3}/;
console.log(expresion.test('ad3rf1hj4')); //Escribe true
console.log(expresion.test('a3f1j4')); //Escribe false
console.log(expresion.test('ab3fg1')); //Escribe false
```

Opciones

La barra vertical "|" permite indicar que se da por válida la expresión de la izquierda o la de la derecha. Ejemplo:

```
let expresion = /([A-Z][0-9]{6})|([0-9][A-Z]{6})$/;
console.log(expresion.test('A123456')); //Escribe true
console.log(expresion.test('1ABCDEF')); //Escribe true
```

Ejemplo de expresión que valida un código postal (formato por 5 número del 00000 al 52999). Los paréntesis ayudan a agrupar esta expresión.

```
let cp1="49345";
let cp2="53345";
let expresion=/^((5[012])|[0-4][0-9]))([0-9]{3})$/;
console.log(expresion.test(cp1)); //Escribe true
console.log(expresion.test(cp2)); //Escribe false
```

Símbolos abreviados

Son símbolos que se usan con el carácter **backslash** y que permite escribir expresiones de forma aún más rápida. Además, funcionan muy bien con Unicode.

SÍMBOLO	SIGNIFICADO
\d	Dígito, vale cualquier dígito numérico
\D	Cualquier carácter salvo los dígitos numéricos
\s	Espacio en blanco
\S	Cualquier carácter excepto el espacio en blanco
\w	Vale cualquier carácter alfanumérico (**w**ord). Es lo mismo que [a-zA-Z0-9]

\W	Cualquier carácter que no sea alfanumérico. Equivalente a [^A-Za-z0-9]
\0	Carácter nulo
\n	Carácter de nueva línea
\t	Carácter tabulador
\\	El propio símbolo \
\"	Comillas dobles
\’	Comillas simples
\c	Permite representar el carácter c cuando este sea un carácter que de otra manera no sea representable (como [,],/,\,...). Por ejemplo \\ es la forma de representar la propia barra invertida.
\ooo	Permite indicar un carácter Unicode mediante su código actual
\xff	Permite indicar un carácter ASCII mediante su código hexadecimal
\uffff	Permite indicar un carácter Unicode mediante su código hexadecimal

4.3.4. PROBLEMAS CON CARACTERES UNICODE

Los símbolos de la tabla anterior permiten abreviar mucho las expresiones y eso permite por ejemplo que cuando pidamos el nombre de usuario de una persona pensemos en esta posible expresión:

```
let usuario = /^[\w] +$/;
console.log(usuario.test('Jorge')); //Escribe true
console.log(usuario.test('kadmin1234')); //Escribe true
console.log(usuario.test('Ibañez')); //Escribe false
```

No da como válido Ibañez por usar el carácter ñ. Es decir, \w funciona con caracteres alfanuméricos del alfabeto inglés. La solución pasa por indicar los caracteres directamente, o bien indicar un rango mediante los códigos hexadecimales Unicode[1].

Por ejemplo, los caracteres en árabe ocupan el rango que va desde 0600-06FF en hexadecimal. Con lo cual, para validar si un string contiene un carácter árabe:

```
let reg = /[\u0600-\u06FF]/;
console.log(reg.test('سماء')); //Escribe true
console.log(reg.test('Hola')); //Escribe false
console.log(reg.test('Hoمla')); //Escribe true
```

Resulta compleja una expresión que admita códigos alfanuméricos de cualquier lengua. Por eso en el estándar **ES2018** se añadió un nuevo símbolo: **\p**. Este símbolo admite indicar entre llaves una

[1] En la dirección https://en.wikipedia.org/wiki/Unicode_block disponemos de los rangos de las diferentes lenguas y tipos de símbolos Unicode

propiedad Unicode. Las propiedades Unicode son extensas y pueden indicar, por ejemplo, pertenencia a un rango (como **Hebrew** para símbolos hebreos) o propiedades de texto (como **Letter** para indicar una letra[2]).

Utilizar este símbolo y otras características avanzadas para manejar adecuadamente los caracteres Unicode, requieren del uso del modificador **u** al final de la expresión.

```
let usuario = /^[\p{Letter}\p{Numer}]+$/u;
console.log(usuario.test('Jorge')); //Escribe true
console.log(usuario.test('kadmin1234')); //Escribe true
console.log(usuario.test('Ibañez')); //Escribe true
```

El problema es que el símbolo **\p** no está disponible en todos los navegadores.

4.3.5. MÉTODO EXEC

Las variables a las que se asigna una expresión regular son en realidad objetos que tienen métodos disponibles. El más importante es **test**, pero hay algunos más.

Nos puede ser de utilidad, por su potencia, el método **exec**. Este método es similar a test en cuanto a que valida una expresión regular en un texto. Pero lo que hace es devolver un array con información sobre cuándo se encontró la expresión en el texto. En este array los elementos son:

- Elemento con índice 0. Es el primer texto encontrado que cumple la expresión.
- Elemento con propiedad **index**. Es un número que indica la primera posición en la que se encontró el texto.
- Elemento con propiedad **input**. Es el texto original
- Se pueden usar subcadenas de búsqueda entre paréntesis y entonces los índices 1,2,... nos devolverán los textos encontrados con las subcadenas.

Como vemos, el objeto devuelto es complejo, pero permite acciones muy impresionantes:

```
let texto="Este es un texto de prueba";
let expresion=/\w+/;
let res=expresion.exec(texto);
console.log(res);
```

Resultado:

```
▼['Este', index: 0, input: 'Este es un texto de prueba', groups: undefined]
    0: "Este"
    groups: undefined
    index: 0
    input: "Este es un texto de prueba"
    length: 1
```

El primer elemento (**res[0]**) muestra el primer texto que cumple la expresión, la cual busca una serie de letras. Por eso es la palabra Este la primera que lo cumple (después viene un espacio en blanco que ya

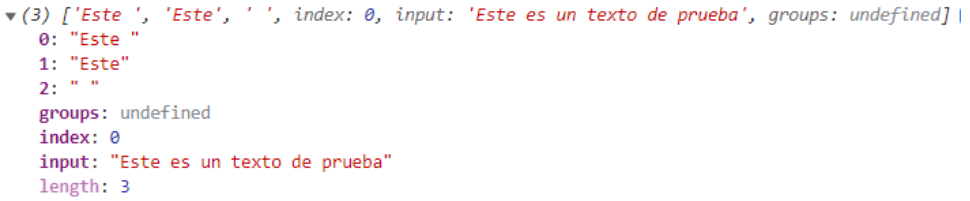
[2] En la dirección https://en.wikipedia.org/wiki/Unicode_character_property se describen las propiedades Unicode, no todas están soportadas en JavaScript

no es una letra). El elemento siguiente: **res.index** o también **res["index"]**, indica el índice del texto en el que empieza el texto que cumple esa expresión. Finalmente, **res.input** escribe el texto tal cual.

Si variamos la expresión:

```
let texto="Este es un texto de prueba";
let expresion=/(\w+)(\s)/;
let res=expresion.exec(texto);
console.log(res);
```

Ahora la expresión es capaz también de buscar subexpresiones gracias a los paréntesis. El resultado es:



Es prácticamente el mismo resultado, pero hay 2 elementos nuevos, el segundo (res[1]) muestra el texto que cumple la primera expresión entre paréntesis (**\w+**) el tercero (res[2]) muestra la primera vez que se cumple la segunda expresión entre paréntesis (**\s**).