

PROGRAMACIÓN DE OBJETOS EN JAVASCRIPT

OBJETIVOS

- Reconocer las ventajas de utilizar objetos en la programación de aplicaciones web
- Identificar las características principales de la programación orientada a objetos en JavaScript
- Utilizar propiedades y métodos de objetos propios o predefinidos
- Crear objetos propios especificando sus propiedades y métodos
- Establecer mecanismos de herencia por prototipado
- Examinar objetos recorriendo sus propiedades y métodos
- Distinguir la utilidad y funcionamiento básico de la notación JSON
- Identificar la utilidad de los objetos predefinidos para manipularlos adecuadamente
- Realizar cálculos avanzados con ayuda del objeto Math
- Utilizar el objeto Date para manipular fechas y horas
- Distinguir el funcionamiento de las expresiones regulares en JavaScript
- Validar expresiones regulares con ayuda de los objetos RegExp

CONTENIDOS

- 6.1 JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS**
 - 6.1.1 ¿QUÉ ES LA POO?
 - 6.1.2 CARACTERÍSTICAS DE LA POO
 - 6.1.3 JAVASCRIPT COMO LENGUAJE ORIENTADO A OBJETOS
- 6.2 USO DE OBJETOS**
 - 6.2.1 ACCESO A PROPIEDADES Y MÉTODOS
 - 6.2.2 OBJETOS LITERALES
 - 6.2.3 OBJETO THIS
 - 6.2.4 RECORRER LAS PROPIEDADES DE UN OBJETO
 - 6.2.5 BORRAR PROPIEDADES DE OBJETOS
- 6.3 USO AVANZADO DE OBJETOS**
 - 6.3.1 CREAR OBJETOS A TRAVÉS DE CONSTRUCTORES
 - 6.3.2 OPERADOR INSTANCEOF
 - 6.3.3 PROTOTIPOS
 - 6.3.4 NOTACIÓN JSON
- 6.4 OBJETOS PREDEFINIDOS**
 - 6.4.1 OBJETO MATH
 - 6.4.2 OBJETO DATE
 - 6.4.3 EXPRESIONES REGULARES
- 6.5 PRÁCTICAS RESUELTA**
- 6.6 PRÁCTICAS RECOMENDADAS**
- 6.7 RESUMEN DE LA UNIDAD**
- 6.8 TEST DE REPASO**

6.1 JAVASCRIPT Y LA PROGRAMACIÓN ORIENTADA A OBJETOS

6.1.1 ¿QUÉ ES LA POO?

Las siglas **POO** se corresponden a **Programación Orientada a Objetos**, un término que ya tiene un largo recorrido dentro de la ciencia de programación de computadoras y que cobró una enorme dimensión a finales de los 80 hasta su total implantación en la década de los 90 del siglo pasado.

En definitiva, la POO es un modelo de programación. Su éxito se debe a que ha conseguido facilitar enormemente la creación de aplicaciones al conseguir que una aplicación sea un conjunto de objetos que se comunican. Cada objeto se programa de forma independiente, consiguiendo una modularización mayor que la propia 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 de la aplicación capaz de recibir mensajes que provocan un procesamiento de dicho de 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 concentraremos 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 y acciones. Los datos son los atributos o propiedades de los objetos. Si por ejemplo, un objeto representara un *coche*, los datos (que se llaman **atributos** o **propiedades**) podrían ser la marca, el color, la potencia, etc. Las acciones (que se llaman **métodos**) podrían ser arrancar, parar, acelerar, repostar,... es decir, lo que un coche es capaz de hacer.

Una vez que un tipo de objeto (en terminología POO un tipo de objeto se conoce como **clase**) se ha definido, se puede utilizar las veces que 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.

6.1.2 CARACTERÍSTICAS DE LA POO

Hay cierto debate sobre si JavaScript es realmente 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. La mayoría de lenguajes define clases y luego crea 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 tener 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.

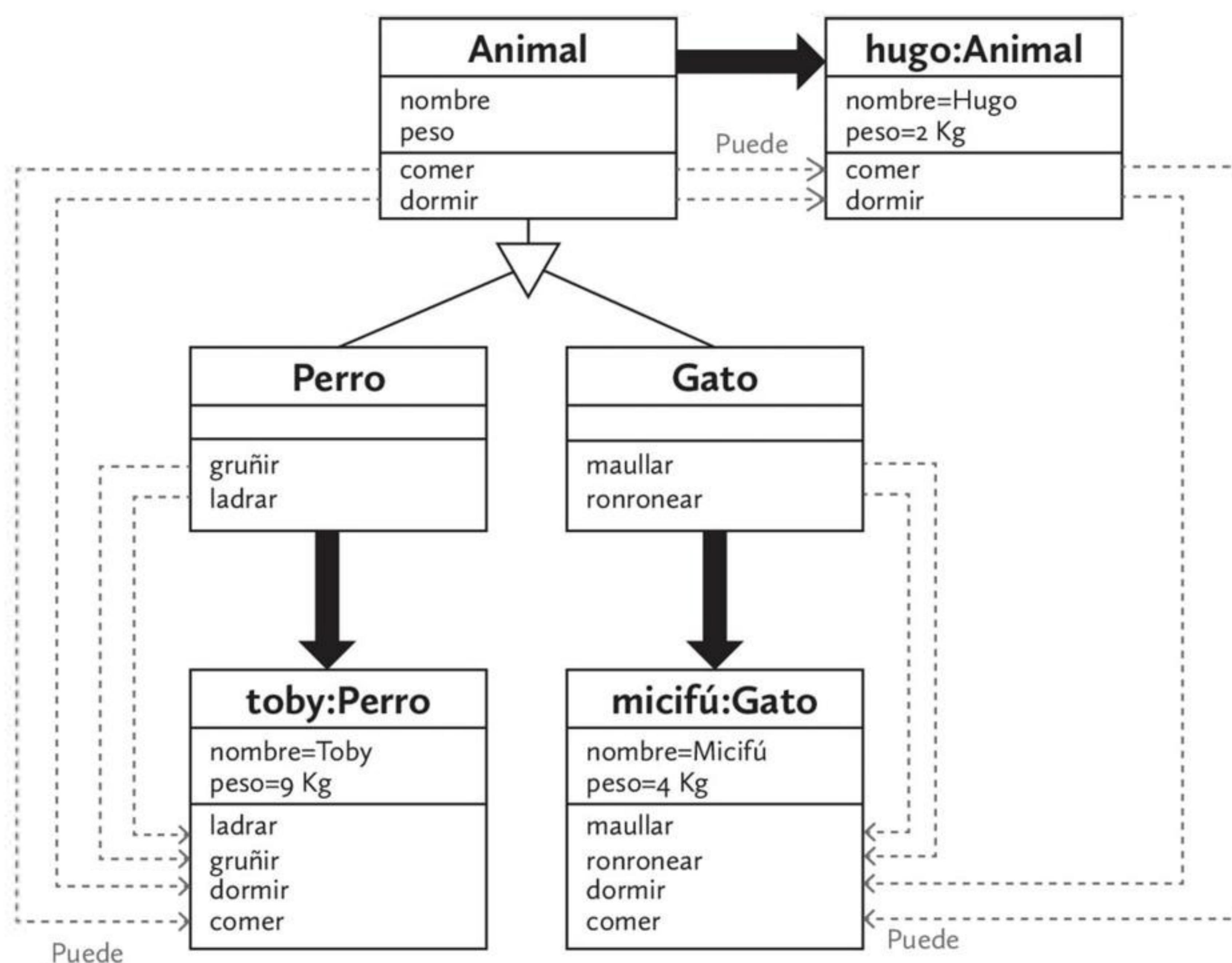


Figura 6.1: Ejemplo de uso de clases y objetos con herencia en la POO

En la Figura 4.1, **Animal**, **Perro** y **Gato** son clases de objetos. Mientras que **toby**, **hugo** y **micifú** son objetos ya concretos. Perro y Gato son clases que descienden de la clase **Animal** (son animales). 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 *micifú*, 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.

6.1.3 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 e 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.

6.2 USO DE OBJETOS

6.2.1 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:

E>b. jeto^propiedad

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

coche.color="Rojo";

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

objeto["propiedad"]

Por ejemplo:

coche["color"]="Rojo";

Para utilizar métodos, la sintaxis es:

objeto.método[parámetros!]

I

Ejemplo:

coche.acelerar(25);

También es posible con:

```
coche["acelerar"]()
```

6.2.2 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", es decir: 10
punto.mostrarCoordenadas=function(){
    return '(${punto.x},${punto.y})';
}
consolé.log(punto.mostrarCoordenadas() ); //Escribe (5,10)
```

La primera línea del código anterior:

```
let punto=new Object();
```

Indica que la variable llamada *punto* es un objeto. Hay una instrucción más sencilla y equivalente a la anterior:

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

Las llaves permiten indicar que la variable punto es un objeto vacío. Tras esa instrucción podemos definir 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})';
    }
};
consolé.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, podemos empezar por métodos, luego definir propiedades, luego definir más métodos, etc. Es decir, el orden lo decidimos a voluntad. Ahora bien, es recomendable, para una mejor legibilidad del código, definir primero las propiedades y luego los métodos.

```
let libro = {
    título : "Manual de UFOlogía",
    "n-serie" : "18702", //Las propiedades pueden llevar espacios,
                    //guiones, etc. por lo que en esos casos se
                    //usan comillas
    autores : [ "Pedro Martínez","Ana León"],
                //Esta propiedad es un array
    editorial : { //La editorial es otro objeto
        nombre : "Inexistente S.A.",
        país : "España"
    }
};
consolé.log(libro.título); //Escribe: Manual de UFOlogía
consolé.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.

6.2.3 OBJETO THIS

En alguno de los códigos de ejemplo anteriores ya hemos utilizado la palabra reservada **this**. Para entender mejor su uso, veamos el siguiente ejemplo de código:

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

El resultado de ese 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 pasa con **y**). Por lo tanto, desde la función que 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})';
    }
};

consolé.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})';
    }
};

consolé.log(punto.mostrarCoordenadas());

```

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

Para entender mejor las posibilidades de *this* veamos este código que es un poco más complejo:

```

function doblarX(){
    this.x*=2;
}

let punto={
    x: 15,
    y:7,
    doble:doblarX
};

let incógnita={
    x: 5,
    dbl:doblarX
};

punto.doble();
incógnita.dbl();
consolé.log(punto.x) //Escribe 30, doble de 15
consolé.log(incógnita.x) //Escribe 10, doble 5

```

La función *doblarX* sirve para multiplicar por dos 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.

6.2.4 RECORRER LAS PROPIEDADES DE UN OBJETO

Para poder recorrer las propiedades de un objeto el bucle idóneo **esfor...in**. Ejemplo:

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

Suponiendo que el objeto punto es el mismo que en los objetos anteriores, se obtendría 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"){
        consolé.log('${prop} tiene el valor ${punto[prop]}');
    }
}
```

6.2.5 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;

consolé.log(objeto.x);      //Escribe 18
consolé.log(objeto.z);      //Escribe undefined
```

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

6.3 USO AVANZADO DE OBJETOS

6.3.1 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 una función constructora.

Los constructores son un elemento muy conocido por los programadores de lenguajes orientados a objetos. En estos lenguajes, estos 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 (en inglés *instances*, que se suele traducir literalmente como instancia, aunque quizá no es la traducción ideal).

```
//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);

consolé.log(a.mostrarCoordenadas());
consolé.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 encarga la palabra **this**, que es la que hace posible llegar a las propiedades

concretas del objeto que se está creando, sabiendo que el valor 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 dos objetos distintos (*a* y *b*) que usan esa función gracias al operador **new** para crear dos objetos distintos, 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 objetos (evitamos usar la palabra clase, que en JavaScript es controvertida) mientras que *a* y *b* son objetos de ese 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:

```
consolé.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.

6.3.2 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:

```
consolé.log(b instanceof Punto); //Devuelve true, b es un Punto
consolé.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** (son objetos). 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 del 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:

```
consolé.log(b. constructor . ríame); //Escribe Punto
```

6.3.3 PROTOTIPOS

6.3.3.1 IDEA DE PROTOTIPO

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

```
consolé.log(a.proto);
```

Se nos muestra:

```
Punto {}
```

Con ello se nos dice que **a** usa el prototipo de clase **Punto**, y que ese prototipo 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 genérica **Object**. Se usa de esta forma:

```
Object.getPrototypeOf(a)
```

El resultado será el mismo.

6.3.3.2 MODIFICAR PROTOTIPOS

Para modificar prototipos basta con indicar la propiedad **prototype** y después definir propiedades y métodos a voluntad. Esta propiedad, como es lógico, solo está disponible en las funciones

constructoras (en realidad está disponible sobre cualquier función, pero eso es otra cuestión). Así si escribimos este código:

```
consolé.log(Punto.prototype);
```

Obtendremos el prototipo de la función Punto, que será un objeto vacío porque no hemos definido nada en 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** 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:

```
Punto { sumaXY: [Function], z: 0 }
```

Dará igual que lo hagamos con la expresión Punto, prototype o con (si **p** es un objeto de 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);
consolé.log(a.sumaXY()); //Muestra 30, resultado de sumar 10+20
consolé.log(b.sumaXY()); //Muestra 3, resultado de sumar -3+6
consolé.log(a.z); //Muestra 0
consolé.log(b.z); //Muestra 0
```

Un detalle muy importante es lo 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.

Para aclarar este punto observemos este código:

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

El resultado de este código sería:

```
Punto { x: 10, y: 20, mostrarCoordenadas: [Function], z: 7 }
```

```
Punto { x: -3, y: 6, mostrarCoordenadas: [Function] }
```

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:

```
consolé.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:

```
consolé.log(a.proto.z); //Escribe 0
```

Un detalle muy 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] ;
consolé.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 (para ello usa internamente el método **filter** de los arrays).

6.3.4 NOTACIÓN JSON

JSON es el acrónimo de *JavaScript Objects Notation* (**Notación de Objetos de JavaScript**) que se creó en 2001 por parte de Douglas Crockford con la idea de aprovechar la forma que posee JavaScript de 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 es muy parecido a la forma de crear objetos de JavaScript, se basa en indicar propiedades y valores separados por dos puntos. Los valores se indican igual que en JavaScript (textos entrecomillados, los números tal cual y usando el punto como decimal, etc.) Se permite el uso de arrays, anidar objetos, etc. Pero hay diferencias que hay que reseñar:

- JSON solo admite definir propiedades, no se pueden indicar métodos en el formato JSON.
- En JSON el nombre de las propiedades tienen que ir entre comillas dobles (no se admiten el resto de comillas).

Ejemplo:

```
{
  "título": "Manual de UFOlogía",
  "n-serie": "187C2", //Las propiedades pueden llevar espacios,
                    //guiones, etc. por lo que en esos casos se
                    //usan comí 1 las
  "autores": [      "Pedro Martínez", "Ana León"],
                  //Esta propiedad es un array
  "editorial": { //La editorial es otro objeto
    "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 global llamado **JSON** que permite manipular los datos en este formato.

Este objeto posee dos 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 musical={
  nombre:" Bob",
  apellido:"Dylan",
  fecha_nacimiento: {
    dia:24,
    mes:5,
    año:1941
  },
  discos:['Highway 61 Revisited','Blonde on Blonde','Self Portrait']
}
console.log(JSON.stringify(musical));
```

El resultado:

```
{"nombre":"Bob", "apellido":"Dylan", "fecha_nacimiento":{"día":24,"mes":5,"año":1941}, "discos":["Highway 61 Revisited","Blonde on Blonde","Self Portrait"]}
```

El método **parse** haría lo contrario, a partir del texto crea el objeto. Más adelante veremos las enormes funcionalidades que otorga JSON.

6.4 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 algunos otros que también nos serán útiles.

6.4.1 OBJETO MATH

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

6.4.1.1 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 por ejemplo:

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

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

CONSTANTE	SIGNIFICADO
E	<i>Valor matemático e</i>
LN10	<i>Logaritmo neperiano de 10</i>
LN2	<i>Logaritmo neperiano de 2</i>
LOG10E	<i>Logaritmo decimal de e</i>
LOG2E	<i>Logaritmo binario de e</i>
PI	<i>La constante TT (Pi)</i>
SQRT1_2	<i>Resultado de la división de uno entre la raíz cuadrada de dos</i>
SQRT2	<i>Raíz cuadrada de 2</i>

6.4.1.2 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 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. Por ejemplo, si el número es el 2.3 se redondea a 3.
cos(n)	Coseno de n .
exp(n)	Número e elevado a n . Es decir eⁿ .
floor(n)	Redondea el número n (si es decimal) a su valor inferior. Por ejemplo, si el número es el 2.8 se redondea a 2.
log(n)	Calcula el logaritmo decimal de n .
max(a,b)	Siendo a y b dos números, esta función devuelve el mayor de ellos.
min(a,b)	Siendo a y b dos números, esta función devuelve el menor de ellos.
pow(a,b)	Potencia. Devuelve el resultado de a^b .
random()	Devuelve un número aleatorio, decimal entre cero y uno.
round(n)	Redondea n a su entero más próximo, round(2.3) devuelve 2 y round(2.5) devuelve 3
sin(n)	Devuelve el seno de n .
sqrt(n)	Raíz cuadrada de n .
tan(n)	Tangente de n .

6.4.2 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();
consolé.log(hoy);
```

Escribiría, en la consola de un navegador, algo como:

```
Sat Jun 29 2019 00:58:47 GMT+0200 (Hora de verano romance)
```

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 29 de Junio de 2019 (Sábado) a las 0:58 de la madrugada y, además, se indica que esa fecha está dos horas por encima de la del meridiano de **Greenwich** y que el horario de esa zona en ese 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,dia,hora,miñutos,segundos,ms);
```

Ejemplo:

```
let fecha=new Date(2013,5,27,18,12,0,0);
consolé.log(fecha);
```

Muestra:

```
Thu Jun 27 2013 18:12:00 GMT+0200 (Hora de verano romance) ;
```

Observando el resultado queda patente que el mes 5 es junio (en lugar de mayo), porque se considera que enero es el mes cero.

No es obligatorio indicar todos los datos, podemos indicar solo año, mes y día sin indicar la hora:

```
let fecha=new Date(2013,5,27);
```

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);
consolé.log(fecha2);
```

Saldría:

```
Thu Jan 01 1970 01:00:01 GMT+0100 (Hora estándar romance)
```

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

6.4.2.1 MÉTODOS DE LOS OBJETOS DATE

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

```
var fecha=new Date();
consolé.log( fecha . getFullYear() ); //Escribe el año actual (p.ej. 2019)
```

La lista completa de métodos es la siguiente:

MÉTODO	SIGNIFICADO
getDate()	Devuelve el día del mes de la fecha (de 1 a 31).
getUTCDate()	Devuelve el día del mes de la fecha en formato universal.
getDay()	Obtiene el día de la semana de la fecha (de 0 a 6).
getUTCDay()	Obtiene el día de la semana de la fecha universal.

MÉTODO	SIGNIFICADO
getFullYear()	Obtiene el año (con cuatro cifras).
getUTCFullYear()	Obtiene el año (con cuatro cifras). La diferencia es que UTC usa la fecha global (lo que corresponda al meridiano de Greenwich).
getHours()	Obtiene la hora de la fecha (número que va de 0 a 23).
getUTCHours()	Obtiene la hora en formato universal.
getMilliseconds()	Obtiene los milisegundos en la fecha actual.
getUTCMilliseconds()	Obtiene los milisegundos en la fecha actual pasada al formato universal.
getMinutes()	Obtiene los minutos.
getUTCMinutes()	Obtiene los minutos en formato universal.
getMonth()	Obtiene el número de mes; de 0, enero, a 11, diciembre.
getUTCMonth()	Obtiene el número de mes en formato universal.
getSeconds()	Obtiene la parte de los segundos de la fecha.
getUTCSeconds()	Segundos en formato universal.
getTime()	Obtiene el valor en milisegundos de la fecha. Número de segundos transcurridos desde el 1 de enero de 1970, respecto a esa fecha.
getTimezoneOffset()	Obtiene el valor en milisegundos usando el formato universal.
setDate(día)	Modifica el día del mes de la fecha. Usa el indicado en el parámetro.
setUTCDate(día)	Versión en formato universal del método anterior.
setFullYear(año)	Modifica el año de la fecha.
setUTCFullYear(año)	Versión en formato universal del método anterior.
setHours(hora)	Modifica la hora.
setUTCHours(hora)	Versión en formato universal del método anterior.
setMilliseconds(ms)	Modifica los milisegundos.
setUTCMilliseconds(ms)	Versión en formato universal del método anterior.
setMinutes(miñutos)	Modifica los minutos.
setUTCMinutes(miñutos)	Versión en formato universal del método anterior.
setMonth(mes)	Modifica el número de mes.
setUTCMonth(mes)	Versión en formato universal del método anterior.
setSeconds(segundos)	Modifica los segundos.
setUTCSeconds(seg)	Versión universal.

MÉTODO	SIGNIFICADO
setTime(mi1 isegundos)	Modifica la fecha haciendo que valga la fecha correspondiente a aplicar el número de milisegundos indicados, contados a partir del 1 de enero de 1970.
toString()	Muestra la fecha en un formato más humano de lectura. Ejemplo: <pre>consolé.log(fecha.toDateString());</pre> Muestra: <pre>Mon Aug 27 2019</pre>
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: <pre>var d=new Date(); consolé.log(d.toISOString());</pre> Obtiene, por ejemplo (si son las 1:18:40,268 segundos del día 27 de mayo de 2019 en el meridiano de Greenwich): <pre>2019-05-27T01:18:40.268Z</pre>
toLocaleString([codigoLocal[,opciones]])	Sin parámetros muestra la fecha y hora en formato de texto usando la configuración local. consolé. log(fecha . toLocaleString()); Obtiene, por ejemplo: <pre>27/5/2019 17:02:55</pre> Pero es posible indicar el código de país: consolé. log(fecha . toLocaleString("en")); Ahora muestra: <pre>2019/5/27 5:02:55 PM</pre> Las opciones permite indicar formato numérico, zona horaria, formato de hora, etc. Ejemplo: <pre>consolé.log(fecha.toLocaleString("es", {"timeZone":"UTC"}));</pre>

MÉTODO	SIGNIFICADO
toLocaleDateString()	Muestra la fecha (sin la hora) en formato de texto usando la configuración local.
toString()	Muestra la hora (sin la fecha) en formato de texto usando la configuración local.
toUTCString()	Muestra la fecha en formato de texto usando la configuración habitual de JavaScript.
toJSON()	Versión en formato universal de la anterior.

6.4.2.2 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.

6.4.3 EXPRESIONES REGULARES

6.4.3.1 ¿QUÉ SON LAS EXPRESIONES REGULARES?

La mayoría de 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 en 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 (**Número de Identificación Fiscal**). Este número está formado por 8 números y una letra cuando coincide con el DN1 (**Documento Nacional de Identidad**) de las personas, pero puede haber una letra en la primera cifra en otras circunstancias (si la persona tiene un **Número de Identificación de Extranjero** u otras circunstancias) y esa letra solo puede ser **K L, X, Y o Z**.

Validar todas esas posibilidades, aun sin contar con que la letra final debe cumplir una condición matemática, es muy complejo. Pero una sola expresión regular puede establecer el patrón que cumple el N1F y así facilitar mucho su validación.

6.43.2 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 dos barras de dividir. Entre ambas barras se coloca la expresión regular. 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 N1F.

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 utilizar la otra forma al ser más rápida e incluso entendible.

6.4.33 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/;
consolé.log(expresión.test('casa'))//Escribe true ;
consolé.log(expresión.test('Casa')) //Escribe false
consolé.log(expresión.test('barbacoa'))i //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. La letra *i* minúscula tras el cierre de la expresión indica que no se debe distinguir entre mayúsculas y minúsculas:

```
let expresion=/c/i;
consolé.log(expresión.test('casa')) //Escribe true
```

```
consolé.log(expresión.test('Casa')); //Escribe true
consolé.log(expresión.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/;
consolé.log(expresión.test('casa')); //Escribe true
consolé.log(expresión.test('Casa')); //Escribe true
consolé.log(expresión.test('asador')); //Escribe true
consolé.log(expresión.test('valiosa')); //Escribe false
```

En el código anterior, solo los textos que tengan dentro las letras *ay s* seguidas se consideran válidas. La palabra **valiosa** contiene una *a* y una *s* pero no seguidas, por eso la validación es falsa.

La potencia de las expresiones regulares se dispara con el uso de símbolos especiales. Explicamos 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/;
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresion.test('barbacoa')); //Escribe false
```

Que termine por

El símbolo de dólar (\$) hace que el carácter anterior sea, obligatoriamente, el último del texto obliga a que el siguiente símbolo de la expresión sea el primero que, obligatoriamente, tenga el texto que validemos:

```
let expresion=/a$/;
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresion.test('barbacoa')); //Escribe true
consolé.log(expresion.test('río')); //Escribe false, no acaba por "a"
```

Comodín de cualquier carácter

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

```
let expresion=/a.e/;
consolé.log(expresión.test('caserío')); //Escribe true
consolé.log(expresion.test('aereo')); //Escribe false, debería haber
                                         //un carácter entre "a" y "e"
consolé.log(expresión.test('alambique'));//Escribe false
```

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

Símbolosopcionales

Cuando se pone una serie de símbolos entre corchetes, se permite cualquier de ellos. Por ejemplo, la expresión *I\arft/* la cumpliría cualquier texto con una letra *a* o una *erre* o una *efe* o una *te*. Gracias a ello podemos conseguir códigos como este:

```
let expresión^/[ao]$/;
    //Es válido cualquier texto que acabe por “a” u “o”
consolé.log(expresión.test ('casa')); //Escribe true
consolé.log(expresión.test('barbacoa')); //Escribe true
consolé.log(expresión.test('río')); //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 minúsculas
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresion.test('Casa')); //Escribe false
consolé.log(expresión.test('río')); //Escribe true
consolé.log(expresión.test ('árbol')); //Escribe false
```

Puede sorprendernos que se de como falso el texto *árbol*, pero es que el carácter *á* en Unicode no está entre la letra *a* y la *z*, está mucho después. Por eso hay que usar estas expresiones con cautela. Por ejemplo podríamos cambiar el código por:

```
let expresion=/^a-zAéíóúü]/;
    //Es válido cualquier texto que empiece con minúsculas
consolé.log(expresion.test('casa')); //Escribe true
consolé.log(expresión.test('Casa')); //Escribe false
consolé.log(expresión.test('río')); //Escribe true
consolé.log(expresión.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 sería válida. Ejemplo:

```
let expresion=/^[^AEIOU]/;
consolé.log(expresion.test('Empresa')); //Escribe false
consolé.log(expresion.test('Casa')); //Escribe true
consolé.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 una o más veces
X[*]	la expresión a la izquierda de este símbolo se puede repetir cero o más veces
X?	El carácter a la izquierda de este símbolo se puede repetir cero o una veces
x{<i>n</i>}	Significa que la expresión <i>x</i> aparecerá <i>n</i> veces, siendo <i>n</i> un número entero positivo.
x{<i>n</i>,}	Significa que la expresión <i>x</i> aparecerá <i>n</i> o más veces.
x{<i>m,n</i>}	Significa que la expresión <i>x</i> aparecerá de <i>m</i> a <i>n</i> veces.

Ejemplos:

```
let expresion=/^([AEIOU]).+a/;
consolé.log(expresión.test("Asa")); //Escribe true
consolé.log(expresión.test("Estación")); //Escribe true
consolé.log(expresión.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/;
consolé.log(expresión.test("Asa")); //Escribe true
consolé.log(expresión.test("Estación")); //Escribe true
consolé.log(expresión.test("Ea")); //Escribe true
```

Si quisieramos validar un código postal español, debemos indicar que solo se permiten 5 caracteres numéricos, que tienen que ser eso, exactamente cinco y, por ello, se indican los delimitadores de principio y final.

```
let CPválido=/^([0-9]{5})$/;
```

Paréntesis

Los paréntesis permiten agrupar expresiones. Eso permite realizar expresiones aun más complejas. Ejemplo:

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

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

```
let expresion=/([a-z]{2}[0-9])\{3\}/;
consolé.log(expresión.test("ad3rfjh4")); //Escribe true
consolé.log(expresión.test("a3flj4")); //Escribe false
consolé.log(expresión.test("ab3fgl")); //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])$/;
consolé.log(expresión.test("A123456")); //Escribe true
consolé.log(expresión.test("1ABCDEF")); //Escribe true
```

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

```
let cpl="49345";
let cp2="53345";
let expresion=/^(5[012])|([0-4][0-9])([0-9]{3})$/;
consolé.log(expresión.test(cpl)); //Escribe true
consolé.log(expresión.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 salvo el espacio en blanco
\w	Vale cualquier carácter alfanumérico (<i>word</i>). 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 octal.
\xflf	Permite indicar un carácter ASCII mediante su código hexadecimal.
\uflfff	Permite indicar un carácter Unicode mediante su código hexadecimal.

6.4.3.4 PROBLEMAS CON CARACTERES UNICODE

Los símbolos de la tabla anterior permiten abbreviar 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]+$/;
```

Al aplicarla nos encontramos con una sorpresa:

```
consolé.log(usuario.test("Jorge")); //Escribe true
consolé.log(usuario.test("kadmin1234")); //Escribe true
consolé.log(usuario.test("Orduño")); //Escribe false
```

No da como válido a **Orduño** por usar el carácter **ñ**. Es decir, \w funciona con caracteres alfanuméricos del alfabeto inglés. La solución no es fácil, pasa por indicar los caracteres directamente, o bien indicar un rango mediante los códigos hexadecimales Unicode¹.

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

```
let reg=/[\u0600-\u06FF]/;
consolé.log(reg.test()); //Escribe true
consolé.log(reg.test("Hola")); //Escribe false
consolé.log(reg.test("Hoola")); //Escribe true
```

Pero volviendo al problema inicial, es 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 propiedad Unicode. Las propiedades Unicode son extensas y pueden indicar, por ejemplo, pertenencia a un rango (como **Hebrew** para los símbolos hebreos) o propiedades del 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{Number}]+$/u;
consolé.log(usuario.test("Jorge"));
consolé.log(usuario.test("kadmin1234"));
consolé.log(usuario.test("Orduño"));
```

Ahora aparecerá la palabra **true** tres veces.

El problema es que este símbolo (\p) no está disponible en todos los navegadores. En el momento de escribir este texto Chrome ya lo reconoce, pero no así Firefox y otros navegadores, ya que la norma es muy reciente. Indudablemente en el futuro será adoptado por todos los navegadores.

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

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

6.43.5 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** que ya le hemos utilizado en los ejemplos anteriores, 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 devolverían 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);
consolé.log(res);
```

Resultado:

```
[ 'Este',
  index: 0,
  input: 'Este es un texto de prueba',
  groups: undefmed ]
```

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 no es una letra). El elemento siguiente: **res. index** o también: **res ["índex"]**, 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);
consolé.log(res);
```

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

```
[ 'Este ',
  'Este' ]
```

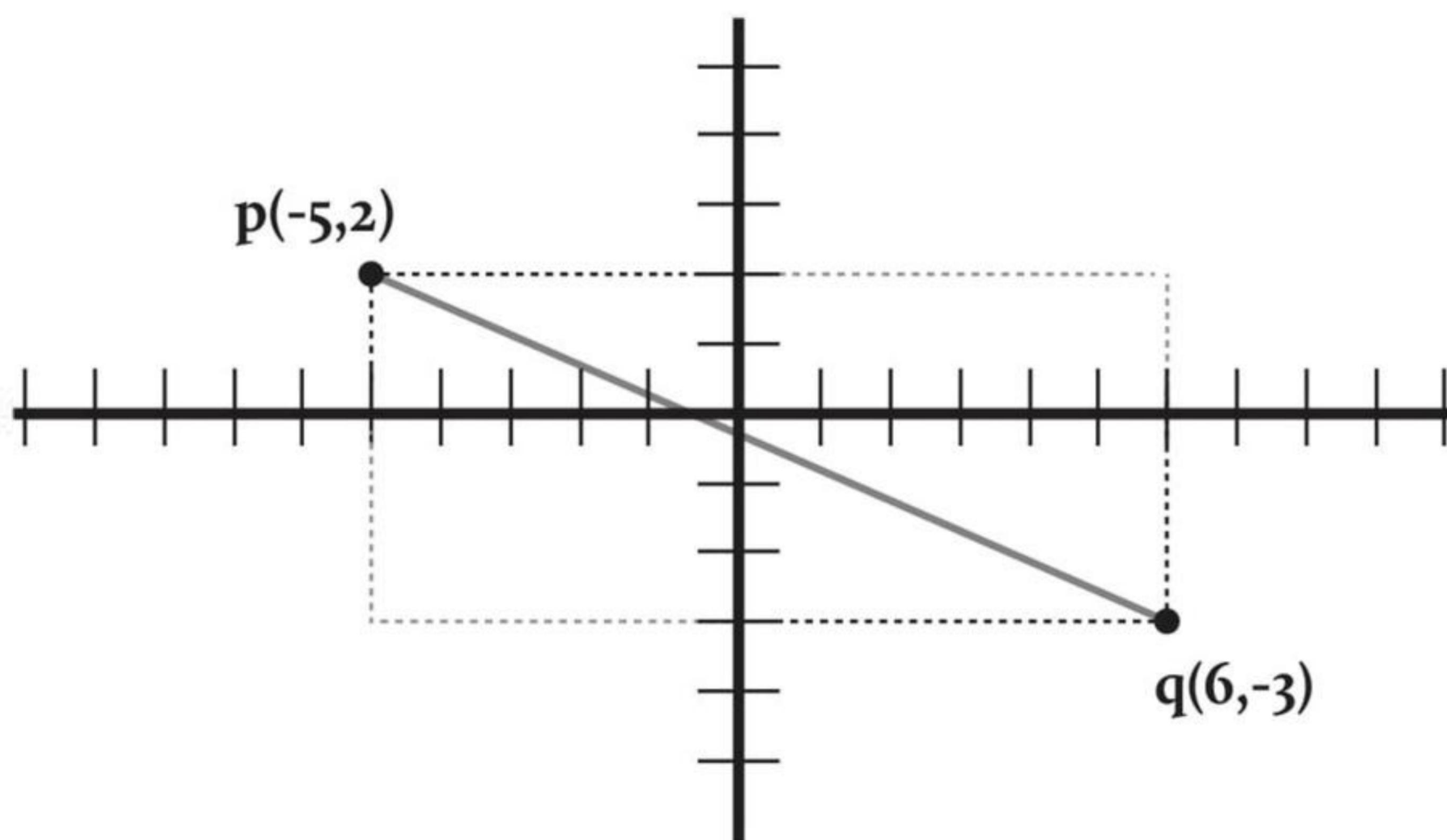
Es prácticamente el mismo resultado, pero hay dos elementos nuevos, el segundo (`res [1]`) muestra el texto que cumple la primera expresión entre paréntesis (`\w+`) el tercero (`res [2]`) muestra la primera vez que se cumple la segunda expresión entre paréntesis (`\s`).

Con pericia, las posibilidades de este método son impresionantes.

6.5 PRÁCTICAS RESUELTAS

Práctica 6.1: Objetos de tipo Punto

- Queremos crear objetos que representen puntos, de ellos necesitamos:
 - Que tengan dos propiedades: **x** y **y**. Servirán para representar las coordenadas del punto.
 - La construcción de los puntos usará una función constructora a la que se le pasan dos números. Si lo que recibe en cada coordenada no es un número, se coloca a cero.
 - Un método llamado **cambiar** al que le pasamos dos números y nos permite modificar las coordenadas del número.
 - Un método llamado **copia** que retorna una copia del objeto.
 - Un objeto llamado **iguales** que recibe un segundo punto y nos dice si ambos puntos son iguales.
 - Un método llamado **suma** que recibe un segundo **punto** y devuelve un tercer punto resultado de sumar las coordenadas de los dos anteriores.
 - Un método llamado **obtenerDistancia** que también recibe un segundo punto y nos devuelve la distancia entre ambos puntos, para esta operación, tener en cuenta:



Para calcular la distancia aplicamos el Teorema de Pitágoras,
la distancia sale de la fórmula:

$$\sqrt{distHoriz^2 + distVert^2} = \sqrt{a^2 + b^2} = \sqrt{6 - (-5)^2 + (-3 - 2)^2} = \sqrt{11^2 + 5^2} = \sqrt{121 + 25} = \sqrt{146} \approx 12,08$$

Figura 6.2: Ejemplo de uso de clases y objetos con herencia en la POO

- Finalmente un método llamado **toString** que retorna un texto con las coordenadas del punto. Por ejemplo (-5,2)

Mostramos la solución solo con código JavaScript. Realmente lo requerido es la función Punto, pero se ha añadido código para probar el correcto funcionamiento de esta función:

```
function Punto(x,y){  
    this.x=x;  
    this.y=y;  
  
    this.cambiar=(x,y)=>{  
        this.x=x;  
        this.y=y;  
    }  
  
    this.copia=()=>(new Punto(this.x,this.y));  
  
    this.suma=(p2)=>(new Punto(this.x+p2.x,this.y+p2.y));  
  
    this.toString=() = >('(${this.x},${this.y})');  
  
    this.obtenerDistancia=function(p2){  
        return Math.sqrt(  
            Math.pow(Math.abs(this.x-p2.x),2) +  
            Math.pow(Math.abs(this.y-p2.y),2)  
        );  
    };  
}  
  
//Prueba de los métodos y construcciones  
var p=new Punto(1,2);  
var q=new Punto(6,-3);  
//modificamos coordendas de p  
p.cambiar(-5,2);  
//r será una copia de p  
var r=p.copia()  
r.x=9;  
//comprobamos que r y p son puntos distintos  
consolé.log("p: "+p.toString());  
consolé.log("r: "+r.toString());  
  
//s es un nuevo punto que toma la suma de coordenadas  
//de p y r  
var s=p.suma(r);  
consolé.log("s: "+s.toString());  
//Obtener distancia entre p y q  
consolé.log("Distancia entre p y q: "+p.obtenerDistancia(q));
```

Práctica 6.2: Añadir método de cálculo de media aritmética a los arrays

- Los objetos de tipo Array (en definitiva, los arrays) poseen numerosos métodos que resultan muy útiles.
- Vemos interesante añadir un método a todos los arrays para el cálculo de la media aritmética de sus elementos.
- Modificar el prototipo de los arrays para añadir dicho método.

SOLUCIÓN: PRÁCTICA 6.2

Bastará con modificar el prototipo del objeto Array. A partir de ese momento todos los arrays en el código podrán calcular la media con el nuevo método. El código muestra la creación del nuevo método y su aplicación en nuevos arrays.

El resultado es francamente fácil gracias a la ayuda de los métodos ya implementados en los arrays:

```
Array.prototype.media=function(){
    let suma=this.reduce((anterior,actual)=>(anterior+actual));
    let tamaño=this.length;
    return suma/tamaño;
}

let al=[1,2,3,4,5,6];
consolé.log(al.media()); //Escribe 3.5
let a2=[10,13,16,21,28,19,45,32,20];
consolé.log(a2. media()); //Escribe 22.666666666666668
```

Práctica 6.3: Objeto para fracciones y añadir métodos para el mcd y el mcm en la clase Math

- Queremos crear objetos que representen fracciones.
- Algunos métodos de estos objetos usan las operaciones matemáticas de cálculo de máximo común divisor y mínimo común múltiplo. Debemos crear estas funciones, pero mejor si son métodos estáticos del propio objeto **Math** (al igual que sus otros métodos vistos en esta misma unidad).
- El máximo común divisor de dos números naturales es el mayor número por el que podemos dividir de forma exacta a ambos números. El mínimo común múltiplo es el número múltiplo más pequeño de ambos números a la vez. Para ambos cálculos podemos usar estos detalles que nos van a facilitar la implementación de ambos métodos:

$$\text{mcm}(a,b) = a * b / \text{mcd}(a,b)$$

$\text{mcd}(a,b) = b$ si el resto de a entre b es cero

$\text{mcd}(a,b) = \text{mcd}(b, a \% b)$ si el resto de a entre b no es cero

- Una fracción consta de un numerador y un denominador, ambos son números enteros. La idea es que cuando se usen estos objetos no se permita modificar directamente el numerador y el denominador, sino que podremos hacerlo mediante métodos.
- El método **getNumerador** obtendrá el valor del numerador, **getDenominador** obtendrá el denominador.
- Los métodos **setNumerador** y **setDenominador** permitirán cambiar tanto el numerador como el denominador. Ambos requieren del nuevo valor.
- El método **cambiarFracción** permitirá cambiar a la vez numerador y denominador.
- Implementar un método llamado **toString** que permita retornar la fracción en el formato **numerador/denominador** (ejemplo 5/6).
- Implementar un método llamado **simplificar** que permita simplificar la función. Eso se consigue calculando el **mcd** del numerador y el denominador. Y dividiendo el numerador y el denominador por ese mcd. Ejemplo:

$$\begin{array}{r} 60 \\ 24 \end{array} = \frac{\cancel{60}/\cancel{12}}{\cancel{24}/\cancel{12}} = \frac{5}{2}$$

mcd

Figura 6.3: Simplificación de fracción

- Implementar un método para la **suma** de fracciones que requiere del cálculo del mínimo común múltiplo de los denominadores. Ejemplo:

$$\begin{array}{r} 3 \\ 10 \end{array} + \begin{array}{r} 6 \\ 4 \end{array} = \frac{(20/10)3 + (20/4)6}{20} = \frac{36}{20} = \frac{9}{5}$$

mcm

Figura 6.4: Suma de fracciones

El mcm de los denominadores es 20, se divide este número entre cada denominador y se multiplica por cada numerador. Se suman los numeradores multiplicados y finalmente se simplifica la fracción.

El resultado debe de ser una nueva fracción con la suma.

- Crear método para la **resta** de fracciones. Se hace igual que la suma pero restando los numeradores.
- Método para la **multiplicación** de fracciones. Se multiplican los numeradores y los denominadores entre sí.
- Método para la **división** de fracciones. Se multiplica de forma cruzada el numerador y el denominador.
- En todas las operaciones matemáticas, se debe de simplificar la fracción resultante.

SOLUCIÓN: PRÁCTICA 6.3

Esta es una práctica ya más compleja por la cantidad de operaciones. La forma de abordarla es ir solventando cada método y probando que funciona bien.

La solución la dejaremos en el mismo archivo, pero la mostraremos por partes. En primer lugar, se realizan los métodos para cálculo de máximo común divisor (**mcd**) y de mínimo común múltiplo (**mcm**). Ambos serán métodos accesibles desde el objeto **Math**. En este caso, no los añadimos al prototipo porque Math no se ha creado para crear objetos a partir de él (no permite *instanciar*). De hecho, no hay función constructora, luego no hay objetos de tipo **Math**.

Usar métodos desde el propio tipo de objeto es utilizar lo que se conoce habitualmente como **métodos estáticos**. Pero, en realidad JavaScript no maneja métodos estáticos de la manera que los manejan lenguajes como Java. Es un artificio conceptual porque la realidad es que Math es un objeto y no una clase. Pero de forma conceptual se manejan este tipo de métodos de la misma forma.

Toda esta explicación teórica nos confundirá. Pero la primera idea es crear ambos métodos asignándolos a Math y probaremos que ambos funcionan:

```
Math.mcd=function(a,b){
    //debemos asegurarnos de que tenemos dos números
    //naturales, de otro modo devolvemos NaN
    //si hay decimales, los quitamos
    a=parseInt(Number(a));
    b=parseInt(Number(b));
    if(isNaN(a) II isNaN(b)) {
        return NaN;
    }
    //si algún número es cero, devolvemos cero
    else if(a==0 || b==0) return 0;
    //tomamos su valor positivo por si acaso
    a=Math.abs(a);
    b=Math.abs(b);

    //Algoritmo
    if(a%b!=0) return Math.mcd(b,a%b);
    else return b;
```

```
}
```

```
Math.mcm=function(a,b){
    //eliminamos errores de la misma forma que en el mcd
    arparseInt(Number(a));
    b=parselnt(Number(b));
    if(isNaN(a) || isNaN(b)) {
        return NaN;
    }
    else if(a==0 || b==0) return 0;
    a=Math.abs(a);
    b=Math.abs(b);

    //resultado basado en el mcd creado antes
    return (a*b)/Math.mcd(a,b);
}
```

Para hacer pruebas podemos escribir este código

```
consolé.log(Math.mcd(60,24)); //Escribe 12
consolé.log(Math.mcm(60,24)); //Escribe 120
```

Una vez comprobado el funcionamiento (convendría hacer más comprobaciones). Este podría ser el código de la función constructora de fracciones:

```
function Fracción(numerador,denominador){
    //puesto que el numerador y denominador son privados
    //se declararan como variables de la función constructora
    var _numerador=numerador;
    var .denominador=denominador;
    //si no se han recibido números, crearemos fracciones nulas
    if(isNaN(parseInt(Number(numerador)))) return null;
    if(isNaN(parseInt(Number(denominador)))) return null;

    //Los métodos sí serán públicos
    this.getNumerador=()=>_numerador;

    this.getDenominador=()=>.denominador;

    this.setNumerador=function(numerador){
        //solo cambiamos el numerador si el número es válido
        if(isNaN(parseInt(Number(numerador)))==false)
            _numerador=parselnt(numerador);//quitamos decimales
    };

    this.setDenominador=function(denominador){
        //solo cambiamos el denominador si el número es válido
    }
```

```

if(isNaN(parseInt(Number(denominador)))==false)
    _denominador=parseInt(denominador);//quitamos decimales
};

this.cambiarFraccion=function(numerador,denominador){
    //aprovechamos los métodos anteriores
    this.setNumerador(numerador);
    this.setDenominador(denominador);
}

this.toString=()=>(_numerador+"+"+_denominador);

this.simplificar=function(){
    //si el numerador o denominador son cero, no simplificamos:
    if(.numerador!=0 & .denominador!=0){
        //miramos los signos
        if(Math.sign(.numerador)==Math.sign(.denominador)){
            // signos iguales ambos números les pasamos a positivos
            _numerador=Math.abs(_numerador);
            .denominador=Math.abs(_denominador);
        }
        else{
            //con signos distintos, dejamos el numerador en negativo si no lo está
            if(.numerador>=0) {
                _numerador=-_numerador;
                _denominador=-_denominador;
            }
        }
        let mcd=Math.mcd(.numerador,.denominador);
        _numerador/=mcd;
        _denominador/=mcd;
    }
}
this.suma=function(f){
    let mcm=Math.mcm(_denominador,f.getDenominador());
    let denominadorRes=mcm;
    let numeradorRes=(mcm/_denominador)*_numerador +
                    (mcm/f.getDenominador())*f.getNumerador();
    let res=new Fraccion(numeradorRes,denominadorRes);
    res.simplificar();
    return res;
}
this.resta=function(f){
    let mcm=Math.mcm(_denominador,f.getDenominador());
    let denominadorRes=mcm;

```

```

let numeradorRes=(mcm/_denominador)*_numerador -
    (mcm/f.getDenominador())*f.getNumerador();
let res=new Fraccion(numeradorRes,denominadorRes);
res.simplificar();
return res;
}

this.multiplica=function(f){
    let res=new Fraccion(
        _numerador*f.getNumerador(),
        _denominador*f.getDenominador()
    );
    res.simplificar();
    return res;
}

this.divide=function(f){
    let res= new Fraccion(
        (_numerador*f.getDenominador()),
        (_denominador*f.getNumerador())
    );
    res.simplificar();
    return res;
}

}

```

Pruebas:

```

var f1=new Fraccion(6,10);
var f2=new Fraccion(6,4);
var f3=new Fraccion(60,24);
f1.setNumerador(3);
consolé.log(f1.toString()); //Escribe 3/10
consolé.log(f2.toString()); //Escribe 6/4
consolé.log(f3.toString()); //Escribe 60/24
f3.simplificar();
consolé.log(f3.toString()); //Escribe 5/2
var fSuma=f1.suma(f2);
var fResta=f1.resta(f2);
var fMultiplica=f1.multiplica(f2);
var fDivide=f1.divide(f2);
consolé.log(fSuma.toString()); //Escribe 9/5
consolé.log(fResta.toString()); //Escribe -6/5
consolé.log(fMultiplica.toString()); //Escribe 9/20
consolé.log(fDivide.toString()); //Escribe 1/5

```

Práctica 6.4: Herencia

- Crea un tipo de objeto que sirva para representar figuras
- Sus propiedades son:
 - Marca, un texto
 - Modelo, un texto
 - Memoria RAM, un número que indica GB de capacidad
 - Capacidad Disco Duro, un número que indica GB de capacidad
 - Pulgadas de pantalla, un número que indica Pulgadas
- Métodos de los ordenadores
 - `toString`, sirve para obtener en forma de texto los detalles del ordenador
- Al crear un ordenador se pueden indicar todos los valores, pero por defecto (sin ser obligado indicarlos) se toman como valores 17 pulgadas, 512 GB de disco duro y 4 GB de RAM. La marca y el modelo sí son necesarias
- Crear otro tipo de objeto que represente ordenadores portátiles, los cuales heredan todo lo de los ordenadores pero añaden una propiedad llamada autonomía, que es un número que expresa horas. Se construye este objeto igual que los ordenadores, pero pudiendo añadir la autonomía (por defecto, 4 horas). Por defecto, en los portátiles las pulgadas son 12 y el disco duro 256 GB.

SOLUCIÓN: PRÁCTICA 6.4

La herencia es la capacidad de que un objeto (en otros lenguajes se hace con clases) herede las capacidades de otro objeto. En JavaScript no es posible como en otros lenguajes, pero se pueden embellecer los prototipos para que adopten las propiedades de otros objetos. Primero nos concentraremos en el objeto **Ordenador**:

```
function Ordenador(marca,modelo,ram=4,disco=512,pulgadas=17){
  this.marca=marca;
  this.modelo=modelo;
  this.ram=ram;
  this.disco=disco;
  this.pulgadas=pulgadas;
  this.toString=function(){
    return "Marca: "+this.marca+"\n"+
      "Modelo: "+this.modelo+"\n"+
      "RAM: "+this.ram+"GB\n"+
      "Disco duro: "+this.disco+"GB\n"+
      "Pulgadas: "+this.pulgadas+" pulgadas\n";
  }
}
```

Es una función constructora simple. Lo interesante está en el objeto **Portátil**, para poder heredar todo lo realizado en los ordenadores deberemos asignar un nuevo objeto de tipo Ordenador al prototipo (mediante la propiedad `proto`). De esta forma en un momento tendremos heredadas las propiedades y métodos y cada objeto de tipo Portátil dispondrá de ellas.

Por supuesto, las nuevas propiedades se llenan de forma explícita y, además, la función `toString` se actualizará añadiendo una nueva línea.

En el siguiente código se expone la función constructora de los portátiles y código para comprobar la corrección de los tipos creados:

```
function Portati1(marca,modelo,ram=4,disco=256,pulgadas=13,autonomia=4){
    this.proto=new Ordenador(marca,modelo,ram,disco,pulgadas);
    this.autonomia=autonomia;
    this.toString=function(){
        return this.proto.toString() +
            "Autonomía: "+this.autonomia+" horas\n";
    }
}

var ol=new Ordenador("HP","EliteDisplay",8,256,23);
var o2=new Ordenador("Dell 1","Inspiron AI0");
var pl=new Portati1("Apple","Macbook Air",8,128,13,12);
var p2=new Portati1("Acer","Aspire");
consolé.log(ol.toString());
consolé.Iog(o2.toString());
consolé.log(pl.toString());
consolé.Iog(p2.toString());
```

Práctica 6.5: Diferencia entre fechas

- Haz una página que pida una fecha usando la notación, dia/mes/año.
- Si hay fallo en la fecha se pide de nuevo. No se comprobará la corrección de la fecha, pero sí que tenga dos dígitos para el día, dos para el mes y dos para el año.
- Se pedirá una segunda fecha y se validará también.
- Finalmente se crearán dos fechas de tipo Date y se mostrará la diferencia en días entre ellas.

SOLUCIÓN: PRÁCTICA 6.5

Los objetos de tipo Date son los idóneos para este tipo de práctica que además utiliza técnicas de validación. Bastará con tomar el tiempo justo antes de que aparezca el cuadro. El tiempo le tomaremos en formato día/mes/año.

Código JavaScript (deberá ser invocado por una página web):

```
//Obtiene un objeto Date de un texto tipo dd/mm/yyyy
function getFecha(fechaString){
    const VAL_FECHA_G=/(\d{2})\|(\d{2})\|(\d{4})/;
    let res=VAL_FECHA_G.exec(fechaString);
    if(res==null)
        return null;
    else{
        //exec extraerá en un array cada apartado entre
        //paréntesis. Empezando por el elemento 1
        // Así en el 1 tendremos el día, en el 2 el mes
        // y en el 3 el año
        //Los objetos Date requieren pasar este orden al revés
        //Al mes hay que restarle 1
        return new Date(res[3],res[2]-1,res[1]);
    }
}

const VAL_FECHA=/\d{2}\|\d{2}\|\d{4}/;
var fechal;
var fecha2;
do{
    fechal=prompt("Escriba la primera fecha           (formato dd/mm/yyyy)");
}while(VAL_FECHA.test(fechal)==false);

do{
    fecha2=prompt("Escriba la segunda fecha           (formato dd/mm/yyyy)");
}while(VAL_FECHA.test(fecha2)==false);

//getFecha devuelve una fecha, de esa fecha usamos el
//método getTime para obtener los milisegundos desde 1970
//que representa esa fecha
let msFechal=getFecha(fechal).getTime();
let msFecha2=getFecha(fecha2).getTime();
//restamos los milisegundos y les pasamos a días
let diferencia=(msFechal-msFecha2)/(1000*60*60*24);
alert(`La diferencia en días de esa fecha es ${diferencia}`);
```

6.6 PRÁCTICAS RECOMENDADAS

Práctica 6.6: Validar URLs

- Crea una aplicación web que sirva para validar URLs
 - Recordamos que una URL está formada por:
 - Protocolo, seguido de dos puntos y, según el protocolo, por hasta tres barras inclinadas (/), pero puede incluso no haber ninguna.
 - Puede haber un nombre de usuario (no es obligatorio) que constará solo de textos, números y/o guiones. Además, se pueden separar varias palabras con puntos, tipo **jorge.sanchez** por ejemplo).
 - Después del nombre de usuario (si le hay) puede haber dos puntos seguidos de la contraseña. En la contraseña vale cualquier serie de caracteres. Hay que hacer notar que si hay contraseña debe haber nombre de usuario seguido de dos puntos, pero puede haber nombre de usuario y no haber contraseña (los dos puntos no se pondrían tampoco).
 - Si hay nombre de usuario se coloca al final de la contraseña un símbolo de arroba. Si no hay contraseña se coloca detrás del nombre del usuario.
 - Después el nombre de la máquina, cuyos caracteres son los mismos que para el nombre de usuario. Al menos un nombre de máquina consta de dos palabras separadas por un punto, pero puede haber más palabras.
 - Opcionalmente, se pueden poner dos puntos seguidos de un número de puerto. El número de puerto es una serie de cifras numéricas.
 - También opcionalmente puede haber al final una ruta, la cual comenzaría por el símbolo de dividir (/) y luego palabras y símbolos de dividir. Esas palabras pueden tener puntos entre medias.
 - Aunque puede haber cadena de búsqueda al final de todo lo anterior, no complicaremos más la validación. Pero al menos, sí se permitirá (sin ser obligatorio) que al final haya cualquier serie de caracteres si van precedidos por el símbolo literal del cierre de interrogación (?).
-

Práctica 6.7: Validar NIF

- Crea una aplicación web que sirva para validar NIFs indicando si es correcto o no.
- Un NIF consta de una posible primera letra que puede ser una X, una Y o una Z. Puede no haber letra y entonces habrá un número.
- Después habrá siete números más.
- Finalmente habrá una letra que cumple una fórmula.

- La fórmula de la letra consiste en dividir los números del N1F entre 23 y tomar el resto. Cuando la primera cifra es una X, se cambia por el número 0 para hacer este cálculo. La Y se cambia por el número 1 y la Z por el número 2. El resto obtenido se comprueba en esta serie:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
T	R	W	A	G	M	Y	F	P	D	X	B	N	J	Z	S	Q	V	H	L	C	K	E

Práctica 6.8: Acceso prohibido

- Crea, una página web que muestre el texto "Bienvenido" pero si la hora es de 9:00 a 14:00 y es un día que no es ni sábado ni domingo.
- Si no se cumple esa premisa, entonces se mostrará el texto "Página cerrada"

Práctica 6.9: Buscaminas con objetos

- Crea una página que sirva para mostrar el mapa del juego de Buscaminas, según las directrices dadas en la Práctica 5.11 "**Mapa buscaminas**", en la página 198 .
- Soluciona el problema usando un objeto (o un tipo de objetos, mejor dicho) que admitan un constructor al que indicamos el número de filas y columnas y el número de minas a dibujar.
- En la construcción se conseguirá colocar los números de minas en las casillas sin mina.
- Además, habrá un método llamado **dibujar** que será el encargado de dibujar el tablero.
- Otro método llamado **recalcular** será capaz de recalcular el tablero de nuevo. Si a este método no se le pasan parámetros, recalcula usando los parámetros de filas, columnas y minas que se le indicaran anteriormente. Pero el método admite que se cambien dichos parámetros

Práctica 6.10: Modificar objetos Map

- Vimos que los objetos Map asocian claves a valores. Las claves no se pueden repetir, pero los valores sí.
- Crea, para este tipo de objetos, un método llamado **invertirMapa** que devuelva un nuevo objeto de mapa donde las claves son los valores del mapa original. Los valores del nuevo mapa serán arrays que contengan las claves donde ese valor se repetía.
- Por ejemplo, si teníamos un mapa con estas claves:
 - Clave 1: Valor: "Óptimo"
 - Clave 2: Valor: "Notable"
 - Clave 3: Valor: "Notable"

- Clave 4: Valor: "Excelente"
- Clave 5: Valor: "Mejorable"
- Clave 6: Valor: "Excelente"
- Clave 7: Valor: "Notable"

El método será capaz de obtener este otro mapa:

- Clave: "Óptimo", Valor: [1]
 - Clave: "Notable", Valor:[2,3,7]
 - Clave: "Mejorable", Valor:[5]
 - Clave: "Excelente", Valor:[4,6]
-
-

Práctica 6.11: Baraja de cartas

- Crea un tipo de objetos que sirvan para representar **Cartas**. Estos objetos tendrán dos propiedades:
 - **palo**. Que será un número de 1 a 4 (donde 1 significa Oros, 2 Espadas, 3 Bastos y 4 Copas)
 - **valor**. Un número del 1 al 10. (8=sota, 9=caballo, 10=rey)
- Los objetos de este tipo se construyen indicando el palo y el valor. Si hay fallos en los datos, se devuelve un objeto nulo en la creación.
- Las cartas tendrán estos métodos:
 - **darValor**. Que recibe un número de palo y un valor para la carta para, con ellos, modificar la carta. Ante datos incorrectos no cambia nada en la carta.
 - **toString**. Método habitual (y estándar) para devolver en forma de texto entendible el valor de la carta. Por ejemplo: **As de Oros**.
- Además, habrá otro tipo de objeto: **Baraja**. La idea es que represente una baraja de cartas españolas. Tendrá los siguientes detalles:
 - La baraja la forman 40 cartas. Para ello tendrá la propiedad **cartas** que será un array de 40 cartas.
 - Al construir la Baraja se rellenan las cartas en el siguiente orden: por palos y cada palo con las cartas del 1 al 10. No se podrá acceder directamente al array fuera de la función constructora.
 - El método **barajar** permite barajar las cartas. Es decir, desordenarlas de forma aleatoria.
 - El método **toString** permite obtener la baraja en forma de texto para saber cómo están ordenadas las cartas.

6.7 RESUMEN DE LA UNIDAD

- La Programación Orientada a Objetos es un modelo que se utiliza ampliamente en el desarrollo de todo tipo de aplicaciones debido a las ventajas que aporta.
- En este paradigma las aplicaciones se entienden como un conjunto de objetos que se comunican entre sí. Podemos concentrarnos en codificar todos los aspectos de cada objeto de manera independiente y así, concentrarnos en tareas más simples y programar de forma más efectiva.
- Aunque JavaScript no sigue el modelo orientado a clases de otros lenguajes como Java o C++, se considera un lenguaje orientado a objetos, aunque con ciertas peculiaridades y limitaciones en la implementación de este modelo. En todo caso, programar código en JavaScript implica manipular objetos en todo momento.
- El operador punto permite acceder a las propiedades y métodos de un objeto: **coche**, **color**, **coche**, **acelerar (10)**, etc. También es posible usar corchetes indicando la propiedad o método entre comillas: **coche["color"]** , **coche["acelerar"]()**.
- La sintaxis para indicar un objeto vacío puede ser mediante **new Object ()** o bien indicando apertura y cierre de llaves: **{ }**. En todo caso, tras crear el objeto las propiedades y métodos se pueden crear sobre la marcha de esta forma:

```
objeto.nombrePropiedad=valor;
objeto.nombreMétodo=function...
```

- Es posible indicar propiedades y métodos en la declaración si dentro de las llaves usamos la sintaxis siguiente:

```
objeto={
    propiedad1:valor1,
    propiedad2:valor2,
    ■ ■ ■
    métodol: function(...) ,  {...}
    ...
}
```

- La palabra reservada **this** se usa para llegar al objeto en sí que se manipula es fundamental para crear muchas de las funciones y métodos de trabajo con objetos.
- Hay una forma de crear objetos que utiliza una función especial llamada **constructor**, que permite crear objetos que comparten el mismo tipo de objeto. Después (a diferencia de otros lenguajes) cada lenguaje puede personalizar sus métodos y propiedades, por lo que esta forma de crear objetos no es como la creación por clases de otros lenguajes.
- El operador **instanceof** permite saber el tipo de un objeto concreto. Todos los objetos son objetos **Object** y los objetos creados a partir de un constructor harán que **instanceof** también devuelva verdadero con el nombre de su constructor (son de ese tipo).

- JavaScript utiliza un mecanismo llamado **prototipo** para conseguir mecanismos de herencia entre objetos. Cuando varios objetos se crean a partir del mismo constructor, todos ellos comparten un objeto común (sobre el que tienen un enlace) que es el prototipo de esos objetos. A ese prototipo le podemos asignar métodos y propiedades y, entonces, serán métodos y propiedades que todos los objetos compartirán. En realidad hay un prototipo común a todos los objetos que es el objeto **Object**.
- El formato JSON es un formato documental muy relacionado con JavaScript porque usa una notación similar a la de los objetos literales de JavaScript. No permite métodos y el nombre de las propiedades debe ir entrecomillado. Es un formato ideal para almacenar datos y que estos se manipulen desde JavaScript o desde otros lenguajes.
- En JavaScript existen numerosos objetos ya predefinidos. Algunos de los más útiles son:
 - El objeto **Math** que permite realizar operaciones matemáticas avanzadas como: cálculos trigonométricos, potencias, logaritmos, números aleatorios, etc.
 - El objeto **Date** para el manejo de fechas y horas que posee numerosos métodos para realizar las operaciones más habituales.
 - Las expresiones regulares son en realidad objetos **RegExp**. Admiten una sintaxis muy cómoda que permite indicar expresiones regulares entre signos de dividir. A través de las expresiones regulares podemos validar que un texto cumpla unas determinadas y complejas reglas sintácticas.
 - Las expresiones regulares son muy útiles y conviene aprender sus símbolos y posibilidades por el tremendo ahorro y eficiencia a la hora de escribir código.

6.8 TEST DE REPASO

1.- Un objeto llamado personal tiene una propiedad llamada nombre. ¿Cuáles de estas formas son válidas?

- a) `personal.nombre`
- b) `personal["nombre"]`
- c) `personal['nombre']`
- d) `personal[nombre]`
- e) `personal{"nombre"}`
- f) `personal{"nombre"}`
- g) `personal{nombre}`
- h) Ninguna es válida

2.- Queremos crear un objeto llamado dado y en él un método llamado nueva-recolocacion. ¿Cuáles de estas sintaxis son válidas? (el cuerpo de la función le dejamos con puntos suspensivos).

- a) `let dado={`
 `nueva-recolocacion:function()`
 `{.■■■}`
- b) `let dado={`
 `"nueva-recolocacion":`
 `function() {...}`
- c) `let dado={};`
 `dado.nueva-recolocacion=`
 `function() {}`
- d) `let dado={};`
 `dado."nueva-recolocacion"=`
 `function() {}`
- e) `let dado={};`
 `dado["nueva-recolocacion"]=`
 `function() {}`

3.- ¿Qué propiedad de la POO es la que se relaciona con la capacidad de ocultar algunos métodos y propiedades de los objetos?

- a) Herencia
- b) Polimorfismo
- c) Abstracción
- d) Encapsulamiento

4.- ¿Qué propiedad de los objetos permite acceder a su prototipo?

- a) `getPrototype`
- b) `__proto__`
- c) `prototype`
- d) Ningún objeto puede acceder a su prototipo

5.- ¿Qué escribe este código por pantalla?

```
function f(){
    consolé.log(this.nombre);
}

function Persona(nombre){
    this.nombre=nombre,
    this.escribirNombre=f
}

let personal=new
    Persona("Amelia");
let persona2=new
    Persona("Darío");
personal.escribirNombre();
persona2.escribirNombre();

a) f
    f
b) function
    function
c) Amelia
    Darío
d) Ocurre un error: this solo se puede
    usar en métodos de objetos.
```

6.- ¿Puede un objeto modificar las propiedades y métodos que hereda de su prototipo?

- a) Sí
- b) No
- c) Solo las propiedades
- d) Solo los métodos

7.- ¿Qué bucle permite recorrer las propiedades de un objeto?

- a) `for .. in`
- b) `for. .of`
- c) `for`
- d) Todos ellos

- 8.- ¿Se pueden añadir nuevos métodos a las clases predefinidas a través de su prototipo?**
- a) No
 - b) Sí
 - c) Solo en el caso de Arrays, Mapas y Conjuntos
 - d!) Solo para los Arrays y Strings
- 9.- ¿Cuáles de estos elementos no pueden estar en un documento JSON?**
- a) Propiedades sin entrecomillar
 - b) Propiedades con comillas simples
 - c) Propiedades con comillas dobles
 - d) Métodos sin entrecomillar
 - e) Métodos con comillas simples
 - f) Métodos con comillas dobles
- 10.- Queremos convertir un objeto llamado *objl* de JavaScript a su formato JSON correspondiente. ¿Qué instrucción lo consigue?**
- a) *objl.stringify()*
 - b) *objl.parse()*
 - c) *JSON.stringify(objl)*
 - d) *JSON.parse(objl)*
- 11.- ¿Cómo podemos conseguir la raíz cuadrada de 2 en JavaScript?**
- a) *Math.sqrt(2)*
 - b) *Math.SQRT2*
 - c) Con las dos anteriores
 - d) No se puede obtener la raíz cuadrada en JavaScript
- 12.- Queremos crear un objeto de fecha que represente el día 3 de mayo de 2017 ¿Qué expresión lo consigue?**
- a) *let f=new Date(3,5,2017);*
 - b) *let f=new Date(2017,5,3);*
 - c) *let f=new Date(5,3,2017);*
 - d) *let f=new Date("3/5/2017","es");*
- 13.- ¿Qué método permite ver una fecha en formato local de España?**
- a) *toLocaleString("es");*
 - b) *toString("es");*
 - c) Cualquiera de los dos anteriores
 - d) No se puede especificar código de idioma, se debe usar la configuración local del usuario
- 14.- ¿Qué significa el código \D en las expresiones regulares?**
- a) Cualquier código alfanumérico
 - b) Cualquier código alfábético
 - c) Cualquier dígito numérico
 - d) Cualquier carácter no alfanumérico
 - e) Cualquier carácter no alfábético
 - f) Cualquier carácter no numérico
- 15.- ¿Qué textos validarían esta expresión regular?/[0-9] {3}\$/**
- a) *Alarma123*
 - b) *Alarma123\$*
 - c) *123*
 - d) *123\$*
 - e) *123\$Alarma*
 - f) *Alarma123\$Alarma*
- 16.- ¿Qué textos validarían esta expresión regular?/^ [^w] /**
- a) *w*
 - b) *www*
 - c) *labe*
 - d) *abe*
 - e) *^w*
 - f) *^www*
 - g) *^labc*
 - h) *^abc*
 - i) *-w*
 - j) *-www*
 - k) *-abe*
 - l) *-labe*