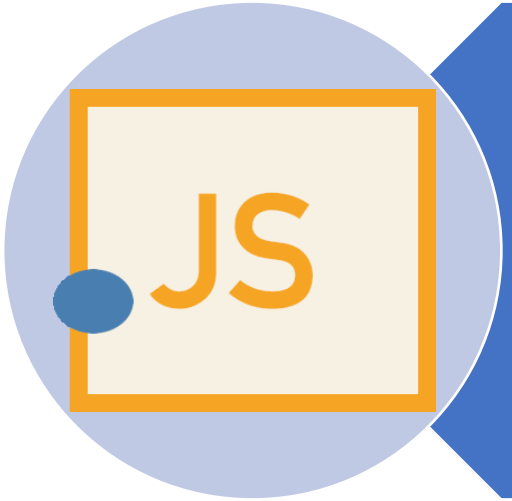
Jorge García Flores

Desarrollo Web en Entorno Cliente

Tema 5



Funciones. Expresiones regulares

funciones

Las funciones junto con los objetos son uno de los principales bloques de construcción de JS.

Una función en JS es parecida a un procedimiento (conjunto de instrucciones que realiza una tarea o calcula un valor), pero las funciones permiten tomar datos de entrada (al empezar la función) y devolver un dato de salida (al finalizar la función).

Las funciones son llamadas e invocadas mediante su nombre cuando queremos que realicen la tarea para las que se han construido.

Para usar una función, se debe definir en algún lugar donde sea visible para llamarla (ámbito).

Para usar una función primero hay que definirla, definición de función (también denominada declaración de función o expresión de función).

La definición de función consta de la palabra clave *`function`*, seguida de:

- El nombre de la función.
- Una lista de parámetros de la función, entre paréntesis y separados por comas (los datos de entrada).
- Las instrucciones JS encerradas entre llaves, *{ }*, que definen lo que hace la función, llamado el cuerpo de la función.

funciones

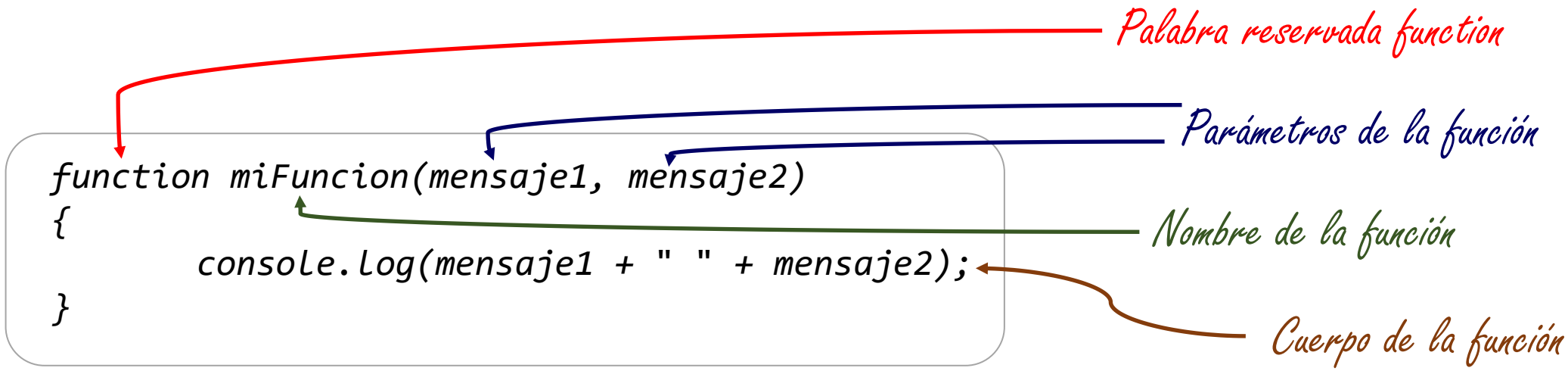
Para llamar a una función, también llamado invocar la función, se utiliza el nombre de la función seguido de paréntesis, ( ), con los argumentos que se le pasan a la función separados por comas y encerrados entre los paréntesis.

Esto provocará que se ejecuten las instrucciones contenidas dentro del cuerpo de la función.

Se llaman **parámetros** a los datos que se le pasa a la función cuando se define.

Y **argumentos** a los datos que se le pasa a la función en las distintas invocaciones de la misma a lo largo del código del script.

```
miFuncion("Hola","Jorge");
```



funciones

Cuando acaban de ejecutarse las instrucciones contenidas dentro del cuerpo de la función o se encuentra con la palabra reservada *return*, la función retorna al punto de código donde fue llamada y, a continuación, se ejecuta la siguiente línea de código del script.

```
miFuncion("Hola","Jorge");
console.log(numero);
```

Llamada a la función, cuando acabe volverá a este punto

Ahora, el script continua con esta instrucción

funciones

Se pueden usar valores por defecto para los parámetros cuando se define la función, estos valores por defecto se usarán en el caso de que no se especifiquen cuando se llame a la función.

Para usar valores por defecto, justo a continuación del parámetro, se pone un signo de igual = seguido del valor por defecto escogido.

```
function miFuncion(mensaje1, mensaje2="!")
{
    console.log(mensaje1 + " " + mensaje2);
}
```

```
miFuncion("Hola");
```

```
Hola !
```

funciones

Se puede devolver un valor, dato de salida, usando la palabra reservada *return* seguido del valor a devolver.

Esto, sin embargo, finaliza en ese punto la función y retorna a la línea desde donde se llamó a la función.

```
function miFuncion(mensaje1, mensaje2="!")
{
    return;

    // lo que sigue jamás se va a ejecutar
    console.log(mensaje1 + " " + mensaje2);
}
```

```
function cuadrado(numero=0)
{
    return numero*numero;
}

console.log(cuadrado(2));
```

funciones

No se puede devolver varios valores a través de *return*. Solo se puede devolver un único dato.

En el caso de querer devolver más de un dato se puede devolver un array formado con los valores que se necesitan retornar; de igual manera, podría devolverse un objeto literal.

Los arrays, arreglos o matrices se estudiarán en el próximo tema.

```
function miFuncion()
{
  const resultado=[1, 2, 'hola'];
  return resultado;
}

const valores= miFuncion();
console.log (valores[0], valores[2]);
```

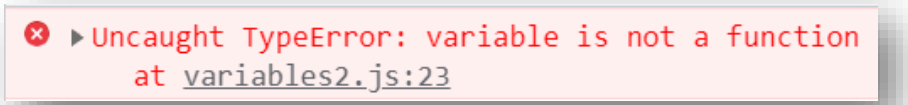
funciones anónimas

Existe otra forma de crear funciones llamada *Function Expression* (expresión de función) o funciones anónimas porque no tienen nombre, pero se asocian a una constante o variable y pueden ser invocadas a través del nombre de dicha constante o variable.

Si se asigna la función a una constante, como es una constante, nos aseguramos de que el nombre de la función no será utilizado para otra cosa distinta más adelante a lo largo del código.

```
const miFuncion = function (mensaje1, mensaje2="!")
{
        console.log(mensaje1 + " " + mensaje2);
}

miFuncion("Hola");
```

```
function variable()
{
  console.log("Hola");
}

var variable=123;

variable();
```



funciones anónimas

La diferencia entre declarar una función y usar una expresión de función reside en el *hoisting*, ya visto cuando se estudiaron las variables.

El concepto de *Hoisting* es una manera general de referirse a cómo funcionan los contextos de ejecución en JS (específicamente las fases de creación y ejecución).

El concepto sugiere que las declaraciones de variables y funciones son físicamente movidas al comienzo del código, pero lo que sucede en realidad es que las declaraciones de variables y funciones son asignadas en memoria durante la fase de compilación, pero quedan exactamente en dónde fueron escritas en el código.

```
saludo();

function saludo()
{
  console.log("Hola");
}
```

En el ejemplo superior, aunque parezca extraño funciona, gracias al *hoisting:*

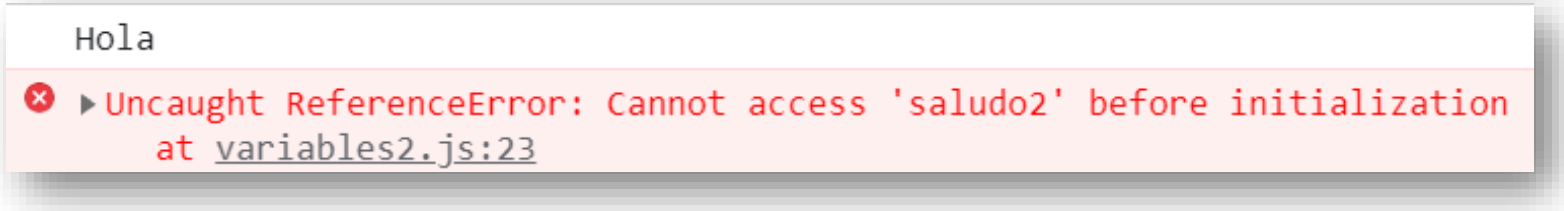
En la primera fase de ejecución del código, el motor JS va a recorrerlo analizándolo, cargando las declaraciones de variables y funciones en memoria.

Por eso, en la fase de ejecución cuando es llama a la función está ya reside en memoria y no da errores.

funciones anónimas

La diferencia es que si la función es llamada antes de declarar una función funcionará, gracias al *hoisting*, pero si es llamada antes de usar una expresión de función dará error, pues el *hoisting* funciona, como ya se ve estudio, con la declaración o definición, pero no con la inicialización.

```
saludo();

function saludo()
{
   console.log("Hola");
}

saludo2();

const saludo2 = function()
{
   console.log("Hola2");
}
```



arrow functions

Existe otra forma de definir funciones llamadas funciones flecha (*Arrow Functions*), esta forma es muy resumida a la hora de escribir código.

Se basan en las expresiones de función, sustituyendo la palabra reservada *function* por el símbolo => pero colocado justo detrás de los paréntesis.

Para llamar a la función se realiza de la misma forma en los tres casos :

```
saludo();
```

```
function saludo(){
        console.log("Hola");
}
```

```
const saludo = function () {
        console.log("Hola");
}
```

```
const saludo = () => {
        console.log("Hola");
}
```

arrow functions

En las funciones flecha (*Arrow Functions*), se puede acortar el código:

Si solo contienen una línea dentro del cuerpo de la función no hace falta colocar las llaves:

```
const saludo = () => console.log("Hola");
```

Además, si solo contienen una línea dentro del cuerpo de la función se da por implícito la palabra *return*:

```
const saludo = () => "Hola";
```

En este ejemplo, la constate *saludo* se le asigna el valor de retorno de la función.

```
const saludo = () =>
{
    console.log("Hola");
}
```

En este caso la cadena de texto "*Hola*" por lo que se podría poner:

```
console.log(saludo());
```

arrow functions

Para definir parámetros en las funciones flecha, se definen en los paréntesis como en la definición normal de funciones.

```
const saludo = (nombre="") => "Hola " + nombre;
```

Cuando solo se le pasa un parámetro a la función los paréntesis son opcionales, siempre y cuando el argumento no tenga definido valores por defecto, y se pueden eliminar.

```
const saludo = nombre => "Hola " + nombre;
```

Las funciones flecha se usan mucho en la actualidad, sobre todo en frameworks, como más tarde se verá en React, ya que facilitan mucho la escritura y comprensión de código. Además, cuando se presenten los arrays se verá algunos usos de funciones flecha con estos elementos, sobre todo a la hora de iterarlos (recorrer sus elementos).

métodos

Las funciones dentro de los objetos se conocen como métodos.

Se definen igual que las expresiones de función, pero dentro del objeto, y se invocan de igual manera que se accedía a las propiedades del objeto, con el operador punto `.` .

```
const coche={

precio: 1000,
conducir: function (){
        console.log("Conduciendo");
        }
};

coche.conducir();
```

Si se quiere acceder a una propiedad del objeto desde dentro de un método se debe usar la palabra reservada *this*.

```
const coche={

precio: 1000,
aumentarPrecio: function (cantidad)
    {
        this.precio +=cantidad;
    }
};

coche.aumentarPrecio(500);
```

métodos

También se puede usar la notación de función flecha para crear los métodos de un objeto.

```
const coche={
precio: 1000,
conducir: function (){
        console.log("Conduciendo");
        },
aumentarPrecio: function (cantidad)
    {
        this.precio +=cantidad;
    }
};

coche.conducir();
coche.aumentarPrecio(5000);
```

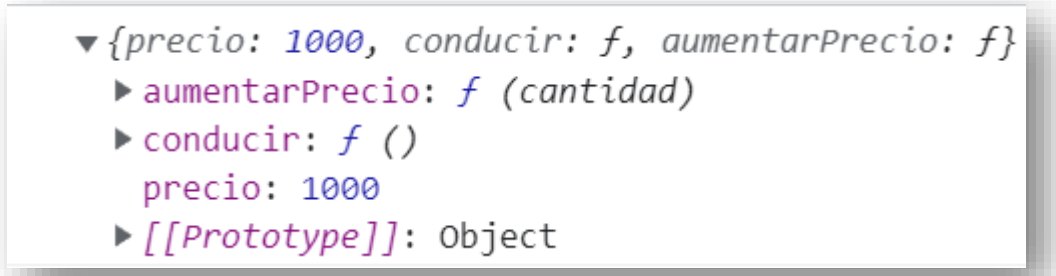
```
const coche={
precio: 1000,
conducir: () => console.log("Conduciendo"),
aumentarPrecio: cantidad => coche.precio +=cantidad,
};

coche.conducir();
coche.aumentarPrecio(5000);
```

En el ejemplo superior, nótese que no se ha utilizado la palabra reservada *this*, ya que haría referencia al objeto *window* y no al objeto *coche*, es por eso que se ha utilizado, en su lugar, *coche.precio*

métodos

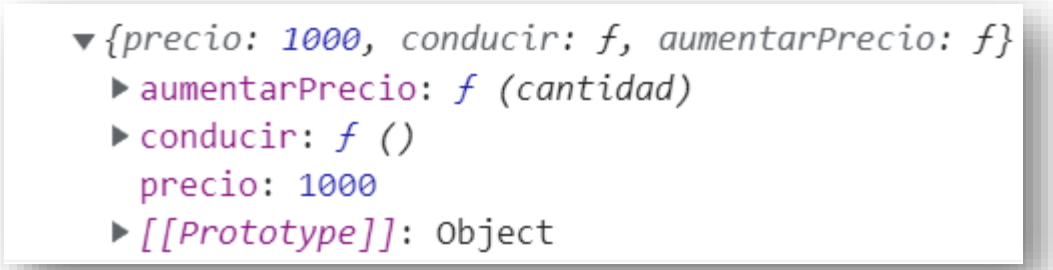
Igual que añadíamos propiedades al objeto una vez creado, podemos añadirle nuevos métodos.

```
const coche={

precio: 1000,
conducir: function (){
        console.log("Conduciendo");
        }
};

coche.aumentarPrecio=function (cantidad){
        this.precio +=cantidad;
};

console.log(coche);
```



métodos

De forma análoga, se pueden añadir a un objeto ya creado nuevos métodos a través de las funciones flecha.




```
const coche={

precio: 1000,
conducir: function (){
        console.log("Conduciendo");
        }
};

coche.aumentarPrecio= cantidad =>coche.precio +=cantidad;

console.log(coche);
```

métodos

Y con la palabra reservada *delete* se pueden eliminar métodos.

```
const coche={

precio: 1000,
conducir: function (){
        console.log("Conduciendo");
        }
};

console.log(coche);
delete coche.conducir;
console.log(coche);
```

métodos

También se puede usar la palabra reservada *delete* con los métodos del objeto definidos por funciones flecha.

```
const coche={
    precio: 1000,
    aumentarPrecio: cantidad =>coche.precio +=cantidad
};

console.log(coche);
delete coche.aumentarPrecio;
console.log(coche);
```

expresiones regulares

Las expresiones regulares (*RegExp* o *RegEx*) son un sistema para buscar, capturar o reemplazar texto utilizando patrones. Estos patrones permiten realizar una búsqueda de texto de una forma relativamente sencilla y abstracta, de forma que abarca una gran cantidad de posibilidades que, de otra forma, sería imposible o conllevaría mucho trabajo y esfuerzo.

Las expresiones regulares también son objetos en JS, y se pueden crear de dos formas:

Usando una expresión regular literal, que consiste en un patrón encerrado entre barras:

```
const expr_regular=/patron/;
```

O llamando al constructor del objeto *RegExp* de la siguiente manera:

```
const expr_regular=new RegExp('patron');
```

expresiones regulares

Las diferencias para usar un método u otro son las siguientes:

- Si la expresión regular va a permanecer constante durante la ejecución del script, no va a cambiar su patrón, se recomienda usar expresiones regulares literales porque el motor de JS compilará la expresión regular para mejorar el rendimiento de la ejecución del script.

- Por otro lado, si el patrón de la expresión regular cambiará durante la ejecución del script o se desconoce el patrón y se obtiene de otra fuente, como la entrada del usuario, una base de datos, etc. se recomienda el uso de constructor para que se realice una compilación en tiempo de ejecución de la expresión regular.

expresiones regulares

Creación	Descripción
`new RegExp(r, flags)`	Crea una nueva expresión regular a partir de r con los *flags* indicados
`/r/flags`	Crea una nueva expresión regular a partir de r , que va entre barras /, con los *flags* indicados

En ambos ejemplos, se establece la expresión regular *jorge*, con un *flag*, la *i*, que indica que no se diferencie entre mayúsculas de minúsculas.

```
// Notación objeto literal
const r = /jorge/i;

// Notación de objeto
const r = new RegExp("jorge", "i");
const r = new RegExp(/jorge/, "i");
```

expresiones regulares

Cada expresión regular creada, tiene unas propiedades definidas, que definen las características de la expresión regular en cuestión.

Además, también tiene unas propiedades de comprobación para saber si un *flag* determinado está activo o no.

Propiedades	Descripción
`.source`	Devuelve un *string* con la expresión regular original al crear el objeto sin *flags*.
`.flags`	Devuelve un string con los *flags* activados en la expresión regular.
`.lastIndex`	Devuelve la posición donde se encontró una ocurrencia en la última búsqueda.
`.global`	Comprueba si el *flag* `g` está activo en la expresión regular.
`.ignoreCase`	Comprueba si el *flag* `i` está activo en la expresión regular.
`.multiline`	Comprueba si el *flag* `m` está activo en la expresión regular.
`.unicode`	Comprueba si el *flag* `u` está activo en la expresión regular.
`.sticky`	Comprueba si el *flag* `y` está activo en la expresión regular.

expresiones regulares

En el siguiente ejemplo, se hace uso de la propiedad *flag*, que contiene un *String* con los *flags* activos en la cadena regular. Como es un *String,* se puede utilizar, mediante *Method Chaining*, los métodos del objeto *String, en concreto* `includes()`, para saber si incluye algún *flag* concreto:

```
const r = /jorge/gi;

console.log(r.source); // 'jorge'
console.log(r.flags); // 'gi'

console.log(r.flags.includes("g")); // true
console.log(r.flags.includes("u")); // false
console.log(r);
console.log(typeof(r));
console.log(typeof(r.flags));
```

```
jorge
gi
true
false
/jorge/gi
object
string
```

expresiones regulares

Las expresiones regulares tienen seis *flags* (indicadores) opcionales que permiten funciones como, por ejemplo, que no distinga entre mayúsculas y minúsculas.

Flag	Booleano	Descripción
`i`	`.ignoreCase`	Ignora mayúsculas y minúsculas.
`g`	`.global`	Búsqueda global. Sigue buscando coincidencias en lugar de pararse al encontrar una.
`m`	`.multiline`	Multilínea. Permite a ^ y $ tratar los finales de línea \r o \n.
`u`	`.unicode`	Unicode. Interpreta el patrón como un código de una secuencia Unicode.
`y`	`.sticky`	*Sticky*. Busca sólo desde la posición indicada por *lastIndex*.

expresiones regulares

En el siguiente ejemplo, se comprueba los *flags* activos con su booleano asociado:

```
const r = /jorge/gi;

console.log(r.global);      // true
console.log(r.ignoreCase);  // true
console.log(r.multiline);   // false
console.log(r.sticky);      // false
console.log(r.unicode);     // false
```

```
true
true
false
false
false
```

expresiones regulares

Los objetos RegExp tienen varios métodos para utilizar expresiones regulares y comprobar si aparecen en el texto, es decir, si el patrón de la expresión regular se encuentra en el texto propuesto.

Método	Descripción
test(str)	Comprueba si en el texto *str* pasado por parámetro se encuentra la expresión regular.
exec(str)	Ejecuta una búsqueda de patrón en el texto *str*. Devuelve un array con las capturas.

Los arrays se estudiarán en el próximo tema.

expresiones regulares

Ejemplo:

```
const r = /jorge/gi;

console.log(r.test("Hola jorge"));                    // true
console.log(r.test("Juan,pedro,jorge,casimiro"));     // true
console.log(r.test("Hola jor ge"));                   // false
console.log(r.test("Adios Jorge!!"));                 // true
```

```
true
true
false
true
```

expresiones regulares

Las expresiones regulares se componen de caracteres simples, como `/jorge/`, o de una combinación de caracteres simples y especiales, como `/ab*c/`.

Las expresiones regulares hechas de caracteres simples permiten crear patrones simples de búsqueda para encontrar coincidencias directas.

Los caracteres especiales permiten hacer búsquedas más complejas. Estos caracteres tienen un significado especial que, en muchos casos, también viene definido por la posición donde se encuentren.

Carácter especial	Descripción
`.`	Comodín, cualquier caracter.
`\`	Invierte el significado de un carácter. Si es especial, lo escapa. Si no, lo vuelve especial.
`\t`	Carácter especial. Tabulador.
`\r`	Carácter especial. Retorno de carro. A menudo denominado CR.
`\n`	Carácter especial. Nueva línea. A menudo denominado «line feed» o LF.

expresiones regulares

Ejemplo:

```
// Buscamos RegExp que encaje con "Manz"
console.log(/M.nz/.test("Manzana"));
console.log(/M.nz/.test("manzana"));
console.log(/M.nz/i.test("manzana"));

// Buscamos RegExp que encaje con "A."
console.log(/A./.test("A."));
console.log(/A./.test("Ab"));
console.log(/A\./.test("A."));
console.log(/A\./.test("Ab"));
```

```
true
false
true
true
true
true
false
```

expresiones regulares

Los corchetes *[ ]* tienen un significado especial, se trata de un mecanismo para englobar un conjunto de caracteres personalizado.

Si se incluye circunflejo ^ antes de los caracteres del corchete, invertimos el significado, pasando a ser “que no exista” el conjunto de caracteres personalizado.

Carácter especial	Descripción
[ ]	Rango de carácteres. Cualquiera de los caracteres del interior de los corchetes.
[^]	No exista cualquiera de los caracteres del interior de los corchetes.
|	Establece una alternativa: lo que está a la izquierda o lo que está a la derecha.

expresiones regulares

Ejemplo:

```
let r = new RegExp("[aeiou]","i");
console.log(r.test("a"));
console.log(r.test("E"));

r = /[^aeiou]/i;
console.log(r.test("a"));
console.log(r.test("E"));
console.log(r.test("T"));
console.log(r.test("m"));

r = /casa|cama/;
console.log(r.test("casa"));
console.log(r.test("cama"));
console.log(r.test("capa"));
```

```
true
true
false
false
true
true
true
true
false
```

expresiones regulares

Dentro de los corchetes se puede establecer dos caracteres separados por guion, por ejemplo *[0-9]*, en este caso se indica el rango de caracteres entre 0 y 9, sin tener que escribirlos todos explícitamente.

De esta forma, se puede crear rangos como *[A-Z]* (mayúsculas) o *[a-z]* (minúsculas), o incluso varios rangos específicos como *[A-Za-z0-9]*.

Carácter especial	Alternativa	Descripción
[0-9]	*\d*	Un dígito del 0 al 9.
[^0-9]	*\D*	No exista un dígito del 0 al 9.
[A-Z]		Letra mayúscula de la A a la Z. Excluye ñ o letras acentuadas.
[a-z]		Letra minúscula de la a a la z. Excluye ñ o letras acentuadas.

expresiones regulares

Carácter especial	Alternativa	Descripción
[A-Za-z0-9]	*\w*	Carácter alfanumérico (letra mayúscula, minúscula o dígito).
[^A-Za-z0-9]	*\W*	No exista carácter alfanumérico (letra mayúscula, minúscula o dígito).
[\t\r\n\f]	*\s*	Carácter de espacio en blanco (espacio, TAB, CR, LF o FF).
[^ \t\r\n\f]	*\S*	No exista carácter de espacio en blanco (espacio, TAB, CR, LF o FF).
	\xN	Carácter hexadecimal número N.
	\uN	Carácter Unicode número N.
	^	Ancla. Delimita el inicio del patrón. Significa **empieza por**.
	$	Ancla. Delimita el final del patrón. Significa **acaba en**.
	\b	Posición de una palabra limitada por espacios, puntuación o inicio/final.
	\B	Opuesta a la anterior. Posición entre 2 caracteres alfanuméricos o no alfanuméricos.

expresiones regulares

Ejemplo:

```
let r = new RegExp("^mas","i");
console.log(r.test("Formas"));
console.log(r.test("Master"));
console.log(r.test("Masticar"));

r = /do$/i;
console.log(r.test("Vívido"));
console.log(r.test("Dominó"));
```

```
false
true
true
true
false
```

```
const r = /fo\b/;
console.log(r.test("Esto es un párrafo de texto."));
console.log(r.test("Esto es un párrafo."));
console.log(r.test("Un círculo es una forma."));
console.log(r.test("Frase que termina en fo"));
```

```
true
true
false
true
```

expresiones regulares

En las expresiones regulares los cuantificadores permiten indicar cuántas veces se puede repetir el carácter inmediatamente anterior.

Existen varios tipos de cuantificadores:

Carácter especial	Descripción
*	El carácter anterior puede aparecer 0 o más veces.
+	El carácter anterior puede aparecer 1 o más veces.
?	El carácter anterior puede aparecer o no aparecer.
{n}	El carácter anterior aparece n veces.
{n,}	El carácter anterior aparece n o más veces.
{n,m}	El carácter anterior aparece de n a m veces.

expresiones regulares

Ejemplo:

```
const r = /a*/;
console.log(r.test(""));
console.log(r.test("a"));
console.log(r.test("aa"));
console.log(r.test("aba"));
console.log(r.test("bbb"));
```

```
true
true
true
true
true
```

```
const r = /a+/;
console.log(r.test(""));
console.log(r.test("a"));
console.log(r.test("aa"));
console.log(r.test("aba"));
console.log(r.test("bbb"));
```

```
false
true
true
true
false
```

expresiones regulares

Una de las características más importantes de las expresiones regulares es lo potente y versátil que resultan las capturas de patrones.

Toda expresión regular que utilice la "*parentización*" (engloba con paréntesis fragmentos de texto) está realizando implícitamente una captura de texto, que es muy útil para obtener rápidamente información.

Carácter especial	Descripción
(x)	El patrón incluido dentro de paréntesis se captura y se guarda en *$1* o sucesivos.
(?:x)	Si se incluye *?:* al inicio del contenido de los paréntesis, se evita capturar ese patrón.
x(?=y)	Busca sólo si *x* está seguido de *y*.
x(?!y)	Busca sólo si *x* no está seguido de *y*.

expresiones regulares

Se deja de utilizar el método *test()* y se utiliza el método *exec()*, que funciona exactamente igual, con la salvedad que devuelve un array (se verán en el próximo tema) con las capturas realizadas.

```
// RegExp que captura palabras de 3 letras.
const r = /\b([a-z]{3})\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre.";
console.log(r.exec(str));
```

```
▼ (2) ['una', 'una', index: 35, input: 'Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre.', groups: undefined] i
    0: "una"
    1: "una"
    groups: undefined
    index: 35
    input: "Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre."
    length: 2
  ▶ [[Prototype]]: Array(0)
```

expresiones regulares

exec() permite ejecutar una búsqueda sobre el texto hasta encontrar una coincidencia. En ese caso, se detiene la búsqueda y devuelve un array con los capturados por la "parentización".

Si el flag *g* está activado se puede volver a ejecutar *exec()* para continuar buscando la siguiente aparición, hasta que no encuentre ninguna más, que devolverá *null*.

```
const r = /\bque\b/gi;
const str = "Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre.";
console.log(r.exec(str));
```

```
▶ ['que', index: 46, input: 'Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre.', groups: undefined]
```

```
console.log(r.exec(str));
```

```
▶ ['que', index: 64, input: 'Hola a todos, amigos míos. Esto es una prueba que permitirá ver que ocurre.', groups: undefined]
```

Bibliografía y recursos online

- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Grammar_and_types
- https://developer.mozilla.org/es/docs/Web/JavaScript/Reference/Functions
- https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Regular_Expressions