1. Que significan las siguientes expresiones regulares y para que pueden servir
   1. /^\s+$/ →uno o más espacios
   2. /^\d{8}[A-Z]$/ →que empiece por 8 dígitos y una letra
   3. /^\d{10}[A-Z]{2}$/ → que empiece por 10 dígitos, 2 letras
   4. /^\d{9}$/ → que empiece y acabe por 9 dígitos
   5. /^\d{3}-\d{3}-\d{3}$/ →que empiece por 3 dígitos, guión, 3 dígitos, guión, 3 digitos
   6. /^\d{3}\s\d{2}\s\d{2}\s\d{2}$/ →que empiece por 3 digitos, espacio, 2 digitos, espacio, 2 digitos, espacio, 2 dígitos
   7. /^\+\d{2,3}\s\d{9}$/ →+, entre 2 / 3 dígitos, espacio, 9 dígitos
2. Definir y comprobar la validez de expresiones regulares que sirvan para comprobar si una cadena de caracteres:
   1. Es un número binario. → **/^(0|[1][01]\*)$/**
   2. Es un teléfono móvil → **^[67]\d{8}$**
   3. Es un teléfono 901, 902 o 903 → **^90[1-3]\d{6}$**
   4. Es daw1, daw2, asir1, asir2, fin1 o fin2 en mayúsculas o minúsculas → /**(daw|asir|fin)[1-2]/gi**
   5. Es una cantidad de dinero que tiene dos decimales (, decimal) y acaba en $ o € →

**^\d\*,\d{2}[$€] → con barra de escape mejor: \$**

**^**(0|[1-9]\d\*),\d{2}[€$]

* 1. Es una página web que comienza con www., acaba con .com o .org o .es y contiene en su nombre “poker”, “porno” o “juego” → **^www\..\*(poker|porno|juego).\*\.(com|org|es)$**
  2. Es una hora que contiene hh:mm:ss (por ej 12:44:21) →

^**([0-1]\d|2[0-3]):[0-5]\d:[0-5]\d$**

* 1. Es una fecha dd/mm/aaaa → **/^(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/\d{4}$/;**
  2. Contraseña de exactamente 6 dígitos →**^\d{6}$**
  3. Contraseña de 6 caracteres con solo dígitos o letras minúsculas → **^(\d|[a-z]){6}$**
  4. Contraseña de entre 6 y 8 caracteres entre los que hay al menos una letra minúscula y un dígito →**/^(?=.\*[a-z])(?=.\*\d)[a-zA-Z\d]{6,8}$/**

**GRUPOS PASIVOS**

1. OPCIONAL

Implementar las siguientes expresiones regulares:

* 1. Solo tienen ‘a’ o solo tienen ‘b’ → **^[a]+$|^[b]+$ → el contrario se hace ^[]**
  2. Solo tengan parejas de ‘a’ y/o parejas de ‘b’ →**(aa|bb)\***
  3. Contienen la cadena ‘aba’ o la cadena ‘bab’ → **(aba|bab)+**
  4. Contienen la cadena ‘ba’ al menos dos veces seguidas → **(ba){2,}**
  5. No tienen más de tres ‘a’ o tres ‘b’ seguidas → **^(?!.\*aaa)(?!.\*bbb).\***
  6. Alternan la ‘a’ y la ‘b’ sin repetirse → **^(ab|ba)\*$**

[**https://extendsclass.com/regex-tester.html?ref=testdev.tools&ref\_type=adv&utm\_campaign=TestDevTools&utm\_medium=web&utm\_source=TestDev.tools**](https://extendsclass.com/regex-tester.html?ref=testdev.tools&ref_type=adv&utm_campaign=TestDevTools&utm_medium=web&utm_source=TestDev.tools)

## 1. Sintaxis básica

* **Límites:**
  + ^: Inicio de la cadena.
  + $: Final de la cadena.
* **Clases de caracteres:**
  + \d: Un dígito (0-9).
  + \w: Cualquier carácter alfanumérico (a-z, A-Z, 0-9, y guion bajo \_).
  + \s: Cualquier espacio en blanco (espacios, tabulaciones, etc.).
  + .: Cualquier carácter, excepto saltos de línea.
* **Negaciones:**
  + \D: No es un dígito.
  + \W: No es un carácter alfanumérico.
  + \S: No es un espacio en blanco.

### 

## 2. Cuantificadores

* **Repeticiones:**
  + \*: Cero o más veces.
  + +: Una o más veces.
  + ?: Cero o una vez.
  + {n}: Exactamente n veces.
  + {n,}: Al menos n veces.
  + {n,m}: Entre n y m veces.

Ejemplo: /a{2,4}/ // 'a' repetida de 2 a 4 veces (aa, aaa, aaaa)

### 3. Grupos y Alternancia

* **Grupos:**
  + (abc): Agrupa varios caracteres como una unidad.
* **Alternancia:**
  + a|b: Coincide con a o con b.

Ejemplo:/(ab|cd)/ // Coincide con "ab" o "cd"

## 4. Clases de caracteres personalizadas

Puedes definir tus propios conjuntos de caracteres usando corchetes []:

* [abc]: Coincide con a, b o c.
* [a-z]: Coincide con cualquier letra minúscula.
* [^abc]: Coincide con cualquier carácter excepto a, b o c (la ^ dentro de los corchetes niega el conjunto).

Ejemplo: /[A-Za-z]/ // Coincide con cualquier letra mayúscula o minúscula

## 5. Grupos de búsqueda anticipada (Lookahead y Lookbehind)

Lookahead (Búsqueda hacia adelante):

* Lookahead positivo (?=...): Verifica que un patrón existe, sin consumir caracteres.
* Lookahead negativo (?!...): Verifica que un patrón no existe.

Lookbehind (Búsqueda hacia atrás) – no soportado por todos los navegadores:

* Lookbehind positivo (?<=...): Verifica que hay un patrón detrás del texto actual.
* Lookbehind negativo (?<!...): Verifica que no hay un patrón detrás.

Ejemplo:

/(?=.\*\d)(?=.\*[a-z])/ // Verifica que hay al menos un dígito y una letra minúscula

### 

## 6. Escape de caracteres especiales

Los caracteres que tienen un significado especial en las expresiones regulares (., \*, +, ?, ^, $, [], (), {}, |, \) deben ser escapados con una barra invertida (\) si quieres buscarlos literalmente.

Ejemplo:

/\./ // Coincide con un punto literal '.'

### 

## 7. Uso en JavaScript

Métodos principales para trabajar con expresiones regulares:

* test(): Comprueba si la expresión regular coincide en una cadena.  
  const regex = /\d+/;

regex.test("abc123"); // true

regex.test("abc"); // false

* match(): Devuelve las coincidencias encontradas o null si no hay coincidencias.  
  "abc123".match(/\d+/); // ['123']
* replace(): Reemplaza el texto que coincide con la expresión regular.  
  "abc123".replace(/\d+/, "456"); // "abc456"
* split(): Divide una cadena en función de una expresión regular.  
  "a,b,c".split(/,/); // ['a', 'b', 'c']

## 8. Modificadores (Flags)

i: Búsqueda insensible a mayúsculas y minúsculas.  
/abc/i // Coincide con "abc", "ABC", etc.

g: Búsqueda global (no se detiene después de la primera coincidencia).  
/\d/g // Busca todos los dígitos en la cadena

## 9. Buena práctica con expresiones regulares

1. **Escapa correctamente los caracteres especiales.**
2. **Combina con métodos de JavaScript** para validar más fácilmente (si una expresión es muy compleja).
3. **Recuerda la eficiencia:** Las expresiones regulares más largas y complejas pueden ser más lentas.

### 

**Ejemplo práctico:**

Validación de una contraseña que debe tener entre 6 y 8 caracteres, al menos una letra minúscula y un número:

const regex = /^(?=.\*[a-z])(?=.\*\d)[a-zA-Z\d]{6,8}$/;

console.log(regex.test("abc123")); // true (válida)

### EXTRA: Tipos de Búsqueda Anticipada (Lookahead):

1. **Búsqueda anticipada positiva** ((?= ...)): Verifica si una parte de la cadena cumple con un patrón, pero **no consume** esos caracteres.
2. **Búsqueda anticipada negativa** ((?! ...)): Verifica que una parte de la cadena **no** cumpla con un patrón, también sin consumir esos caracteres.

### Ejemplo de búsqueda anticipada positiva:

Si tenemos la siguiente expresión:

javascript

Copiar código

/(?=.\*[a-z])(?=.\*\d)[a-zA-Z\d]{6,8}/

Esta expresión busca:

1. (?=.\*[a-z]): Esto es un **lookahead positivo**. Verifica si hay **al menos una letra minúscula** en cualquier parte de la cadena.
   * .\*: Cualquier cantidad de caracteres antes.
   * [a-z]: Al menos una letra minúscula.
2. (?=.\*\d): Otro **lookahead positivo**. Verifica si hay **al menos un número** en cualquier parte de la cadena.
   * .\*: Cualquier cantidad de caracteres antes.
   * \d: Al menos un número.

Después de estas dos verificaciones (que no consumen los caracteres, solo los chequean), la expresión [a-zA-Z\d]{6,8} valida que la cadena tenga entre 6 y 8 caracteres, compuestos de letras y números.

### ¿Cómo funcionan en práctica?

#### Ejemplo con "abc123":

* (?=.\*[a-z]) verifica si **hay una letra minúscula** en cualquier parte de la cadena. Como "abc123" tiene la "a", esta verificación es exitosa.
* (?=.\*\d) verifica si **hay un dígito** en la cadena. "abc123" tiene "1", así que también pasa esta verificación.
* Luego, [a-zA-Z\d]{6,8} asegura que toda la cadena sea de entre 6 y 8 caracteres de longitud y que solo tenga letras y números, lo cual también cumple.

Resultado: ¡válida!

#### Ejemplo con "123456":

* (?=.\*[a-z]) busca una letra minúscula, pero "123456" **no tiene ninguna**, así que falla.
* Aunque (?=.\*\d) encuentra un dígito, como ya falló la verificación de la letra, el resultado final será **inválido**.

### Importante:

* **Grupos pasivos** (lookaheads) **no capturan ni consumen** la parte de la cadena que verifican. Solo se usan para **chequear** si existe (o no) un patrón en la cadena.
* Esto significa que, tras usar un lookahead, la expresión regular continúa su búsqueda desde el mismo punto en la cadena, **sin avanzar**.

### Ejemplo simple de uso:

javascript

Copiar código

const regex = /(?=\d{2})/; // Verifica si hay 2 dígitos seguidos en cualquier parte

console.log("123".match(regex)); // Devuelve "true", ya que "12" está presente

console.log("abc".match(regex)); // Devuelve "false", no hay dígitos

En resumen, los **lookaheads** o **grupos pasivos** son muy útiles cuando necesitas **verificar** que un cierto patrón esté presente en la cadena sin "tocar" o "consumir" esos caracteres directamente.