JavaScript Regular Expressions (Regex)

Regular expressions are patterns used to match character combinations in strings. They are powerful tools for string manipulation, validation, and text processing.

What is a Regular Expression?

A regular expression is a sequence of characters that forms a search pattern. It can be used for:

- Text search and replace
- Validation (email, phone numbers, etc.)
- String parsing and extraction
- Pattern matching

Creating Regular Expressions

1. Literal Notation (Recommended)

Syntax : **const regex = /pattern/flags;**

Here,

1.1 Pattern

The **pattern** is the actual regular expression content between the two forward slashes `/`. It defines what text to match.

Examples of patterns:

```
/hello/        // Matches the exact word "hello"
/[0-9]/        // Matches any digit
/^[A-Z]/       // Matches uppercase letters at start of string
/\w+/          // Matches one or more word characters
```

1.2 Flags (Optional)

Flags are optional modifiers that change how the pattern matching works. They come after the closing slash.

Common flags:

/g // Global search (find all matches, not just first)

/i // Case-insensitive search

/m // Multiline mode (^ and $ match start/end of lines)

/u // Unicode mode (handle Unicode characters properly)

/s // Dotall mode (allows . to match newlines)

/y // Sticky search (matches only from lastIndex position)

2. Constructor Notation

const regex = new RegExp('pattern', 'flags');

1. Basic Patterns

1.1 Anchors tags

- `^` = beginning of string (or line, if multiline `m`)
- `$` = end of string (or line, if `m`)

Exmaples: **`^hello$` means** → *string is exactly "hello"*.

const startRegex = /^abc/; // Starts with 'abc'

const endRegex = /xyz$/; // Ends with 'xyz'

const wordBoundary = /\bword\b/; // Whole word 'word'

1.2 Character Classes

// Literal characters

const helloRegex = /hello/;

// Character sets

const vowelRegex = /[aeiou]/; // Matches any vowel

const notVowelRegex = /[^aeiou]/; // Matches anything except vowels

const rangeRegex = /[a-z]/; // Matches any lowercase letter

const digitRegex = /[0-9]/; // Matches any digit

one example explanation: What `/[^aeiou]/` Means:

- `[ ]` - **Character class**: Match any one character from the set inside
- `^` - **Negation**: When used as the **first character** inside `[ ]`, it means "NOT these characters"
- `[^aeiou]` - Match any single character that is **NOT** a vowel (a, e, i, o, u)

Example Program:

```
const regex = /[^aeiou]/;

console.log(regex.test("a"));  // false - it's a vowel
console.log(regex.test("e"));  // false - it's a vowel
console.log(regex.test("b"));  // true - not a vowel
console.log(regex.test("1"));  // true - not a vowel
console.log(regex.test("!"));  // true - not a vowel
console.log(regex.test(" "));  // true - space is not a vowel
```

output:

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

1.3 Quantifiers

const zeroOrMore = /a*/; // 0 or more 'a's

const oneOrMore = /a+/; // 1 or more 'a's

const zeroOrOne = /a?/; // 0 or 1 'a'

const exactNumber = /a{3}/; // Exactly 3 'a's

const rangeNumber = /a{2,4}/; // 2 to 4 'a's

const atLeast = /a{2,}/; // At least 2 'a's

1.4 Special Characters

const dotRegex = /./; // Matches any character except newline

const wordRegex = /\w/; // Matches word character [a-zA-Z0-9_]

const nonWordRegex = /\W/; // Matches non-word character

const digitRegex = /\d/; // Matches digit [0-9]

const nonDigitRegex = /\D/; // Matches non-digit

const whitespaceRegex = /\s/; // Matches whitespace

const nonWhitespaceRegex = /\S/; // Matches non-whitespace

JavaScript Regex Methods

1. `test()` Method

Purpose: Check if a pattern exists in a string (returns boolean)

Syntax: `regex.test(string)`

Example Program:

```
const regex = /hello/;

console.log(regex.test("hello world"));      // true
console.log(regex.test("world hello"));      // true
console.log(regex.test("HELLO"));           // false (case-sensitive)
console.log(regex.test("goodbye"));         // false

// With flags
const caseInsensitive = /hello/i;
console.log(caseInsensitive.test("HELLO")); // true
```

output:

```
true
true
false
false
true
```

2. `exec()` Method

Purpose: Get detailed match information with capturing groups

Syntax: `regex.exec(string)`

Returns: Array with match details or `null`

Example Program:

```
const regex = /(\w+)\s(\w+)/;
const result = regex.exec("John Doe");

console.log(result[0]);    // "John Doe" (full match)
console.log(result[1]);    // "John" (first group)
console.log(result[2]);    // "Doe" (second group)
console.log(result.index); // 0 (position where match found)
console.log(result.input); // "John Doe" (original string)
```

output:

```
John Doe
John
Doe
0
John Doe
```

In above Example what is this regurla expression /(\w+)\s(\w+)/

1. First Capturing Group: (\w+)

- \w - Matches any **word character** (equivalent to [a-zA-Z0-9_])
  - Letters (a-z, A-Z)
  - Digits (0-9)
  - Underscore (_)
- \+ - **One or more** of the preceding character
- ( ) - **Capturing group** that stores the matched text

2. Whitespace: \s

- \s - Matches any **whitespace character**:
  - Space
  - Tab
  - Newline
  - Other space characters

3. Second Capturing Group: (\w+)

- Same as the first group
- Captures the second word

3. `match()` Method

Purpose: Find matches in a string (called on string, not regex)

Syntax: `string.match(regex)`

Example Program:

```
const text = "The rain in Spain stays mainly in the plain";

// Without global flag (first match only)
console.log(text.match(/ain/));
// ['ain', index: 5, input: 'The rain in Spain...']

// With global flag (all matches)
console.log(text.match(/ain/g));
// ['ain', 'ain', 'ain'] (just the matched text)

// With groups
const dateMatch = "2023-12-25".match(/(\d{4})-(\d{2})-(\d{2})/);
console.log(dateMatch[0]); // "2023-12-25"
console.log(dateMatch[1]); // "2023"
console.log(dateMatch[2]); // "12"
console.log(dateMatch[3]); // "25"
```

output:

```
[
  'ain',
  index: 5,
  input: 'The rain in Spain stays mainly in the plain',
  groups: undefined
]
[ 'ain', 'ain', 'ain', 'ain' ]
2023-12-25
2023
12
25
```

4. `replace()` Method

Purpose: Search and replace patterns in a string

Syntax: `string.replace(regex, replacement)`

Replacement Patterns:

- `$&` - Insert the matched substring
- `$1`, `$2`, ... - Insert captured groups
- `$`` - Insert portion before match
- `$'` - Insert portion after match

Example Program:

```
const text = "Hello world! Hello universe!";

// Simple replacement
console.log(text.replace(/hello/i, "Hi"));
// "Hi world! Hello universe!" (first match only)

// Global replacement
console.log(text.replace(/hello/gi, "Hi"));
// "Hi world! Hi universe!" (all matches)

// Using replacement patterns
console.log("123-456-7890".replace(/(\d{3})-(\d{3})-(\d{4})/, '($1) $2-$3'));
// "(123) 456-7890"

// Using function as replacement
console.log("a1 b2 c3".replace(/\d/g, match => match * 2));
// "a2 b4 c6"
```

output:

```
Hi world! Hello universe!
Hi world! Hi universe!
(123) 456-7890
a2 b4 c6
```

5. `search()` Method

Purpose: Find position of first match (like `indexOf` but with regex)

Syntax: `string.search(regex)`

Example Program:

```
const text = "Hello world! Welcome to JavaScript.";

console.log(text.search(/world/));    // 6
console.log(text.search(/javascript/i)); // 24 (case-insensitive)
console.log(text.search(/python/));   // -1 (not found)

// Useful for checking if pattern exists
if (text.search(/\d/) !== -1) {
    console.log("Contains numbers!");
}
```

output:

```
6
24
-1
```

6. `split()` Method

Purpose: Split string using regex pattern as delimiter

Syntax: `string.split(regex)`

Example Program:

```
const text = "apple,banana;cherry.orange";

// Split by multiple delimiters
console.log(text.split(/[,;.]/));
// ['apple', 'banana', 'cherry', 'orange']

// Split and include delimiters
console.log(text.split(/([,;.])/));
// ['apple', ',', 'banana', ';', 'cherry', '.', 'orange']

// Split by whitespace
console.log("Hello   World\tTest".split(/\s+/));
// ['Hello', 'World', 'Test']

// Split by digits
console.log("a1b22c333d".split(/\d+/));
// ['a', 'b', 'c', 'd']
```

output:

```
[ 'apple', 'banana', 'cherry', 'orange' ]
[
  'apple',  ',',
  'banana', ';',
  'cherry', '.',
  'orange'
]
[ 'Hello', 'World', 'Test' ]
[ 'a', 'b', 'c', 'd' ]
```

Method Comparison Table

Method	Called On	Returns	Best For
`test()`	Regex	Boolean	Quick validation
`exec()`	Regex	Array with details	Detailed match info, capturing groups
`match()`	String	Array	Extracting matches from strings
`replace()`	String	New string	Search and replace operations
`search()`	String	Number (index)	Finding position of pattern
`split()`	String	Array	Splitting strings with complex patterns

When to Use Regular Expressions

1. Validation

```
// Email validation
const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
console.log(emailRegex.test('user@example.com')); // true

// Phone number validation (simple)
const phoneRegex = /^\d{3}-\d{3}-\d{4}$/;
console.log(phoneRegex.test('123-456-7890')); // true

// Password validation (at least 8 chars, 1 uppercase, 1 lowercase, 1 number)
const passwordRegex = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d).{8,}$/;
console.log(passwordRegex.test('Password123')); // true
```

2. Search and Extract

```
const text = "Contact us at: support@example.com or sales@company.org";

// Find all email addresses
const emailPattern = /[^\s@]+@[^\s@]+\.[^\s@]+/g;
const emails = text.match(emailPattern);
console.log(emails); // ['support@example.com', 'sales@company.org']

// Extract dates
const dateText = "Event dates: 2023-12-25, 2024-01-01";
const datePattern = /\d{4}-\d{2}-\d{2}/g;
const dates = dateText.match(datePattern);
console.log(dates); // ['2023-12-25', '2024-01-01']
```

output:

```
 'support@example.com', 'sales@company.org' ]
[ '2023-12-25', '2024-01-01' ]
```

3. Search and Replace

```
const text = "Hello World! hello world!";

// Case-insensitive replacement
const result = text.replace(/hello/gi, 'Hi');
console.log(result); // "Hi World! Hi world!"

// Format phone numbers
const phone = "1234567890";
const formatted = phone.replace(/(\d{3})(\d{3})(\d{4})/, '($1) $2-$3');
console.log(formatted); // "(123) 456-7890"
```

output:

```
Hi World! Hi world!
(123) 456-7890
```

4. Splitting Strings

```
const csv = "apple,banana,cherry,date";
const fruits = csv.split(/,/);
console.log(fruits); // ['apple', 'banana', 'cherry', 'date']

const complexText = "Hello123World456Test";
const parts = complexText.split(/\d+/);
console.log(parts); // ['Hello', 'World', 'Test']
```

output:

```
[ 'apple', 'banana', 'cherry', 'date' ]
[ 'Hello', 'World', 'Test' ]
```

Sample Example Programs:

```
/*1. Basic Pattern: Lowercase to Uppercase Conversion

Regex: /[a-z]/g

Explanation: Matches all lowercase letters

[a-z] - Character class for lowercase letters a to z

g - Global flag (find all matches)
*/
const text = "Hello World 123";
const result = text.replace(/[a-z]/g, match => match.toUpperCase());
console.log(result); // "HELLO WORLD 123"


/*2. Indian Phone Number Validation
Regex: /^(\+91[\-\s]?)?[6-9]\d{9}$/

Explanation:

^ - Start of string

(\+91[\-\s]?)? - Optional country code (+91 with optional hyphen or space)

[6-9] - Must start with 6,7,8, or 9

\d{9} - Exactly 9 more digits

*/
const phoneRegex = /^(\+91[\-\s]?)?[6-9]\d{9}$/;
console.log(phoneRegex.test("9876543210"));    // true
console.log(phoneRegex.test("+91-9876543210")); // true
console.log(phoneRegex.test("2876543210"));    // false (starts with 2)
console.log(phoneRegex.test("987654321"));     // false (only 9 digits)


/* 3. Accept Single Digit
Regex: /^\d$/

Explanation:
^ - Start of string
\d - Exactly one digit (0-9)
$ - End of string
*/
const singleDigit = /^\d$/;
console.log(singleDigit.test("5"));    // true
console.log(singleDigit.test("12"));   // false
console.log(singleDigit.test("a"));    // false
```

```
/*4. Accept Multiple Digits
Regex: /^\d+$/

Explanation:

^ - Start of string

\d+ - One or more digits

$ - End of string
*/
const multipleDigits = /^\d+$/;
console.log(multipleDigits.test("123"));     // true
console.log(multipleDigits.test("0"));       // true
console.log(multipleDigits.test("12a34"));   // false


/*5. Email Validation

Regex: /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/

Explanation:

^[a-zA-Z0-9._%+-]+ - Local part: letters, numbers, ._%+-

@ - Literal @ symbol

[a-zA-Z0-9.-]+ - Domain name: letters, numbers, .-

\. - Literal dot

.[a-zA-Z]{2,}$ - TLD: 2+ letters  (TLD: Top-Level Domain) &  {2,} - Quantifier that
means "2 or more" of the preceding element

Explanation of this Regular Expression /^[a-zA-Z0-9._%+-]
```

The ^ at the beginning suggests this is meant to be part of a larger regex pattern.

The Character Class: [a-zA-Z0-9._%+-]

1. **Letters (Lowercase)** - a-z

- Matches: **a, b, c, ..., z**
- Purpose: **Allows all lowercase English letters**
- Example: **"user", "name", "email"**

2. **Letters (Uppercase)** - A-Z

- Matches: **A, B, C, ..., Z**
- Purpose: **Allows all uppercase English letters**
- Example: **"User", "NAME", "Email"**

3. **Digits** - `0-9`

- Matches: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9**
- Purpose: **Allows numbers**
- Example: **"user123", "test2023", "page1"**

4. **Dot** - `.`

- Matches: **Literal dot character .**
- Purpose: **Allows dots in usernames/emails**
- Example: **"first.last", "john.doe"**

5. **Underscore** - `_`

- Matches: **Literal underscore character _**
- Purpose: **Commonly used in usernames and identifiers**
- Example: **"user_name", "test_value"**

6. **Percent** - `%`

- Matches: **Literal percent character %**
- Purpose: **Used in encoded URLs and special formats**
- Example: **"discount%", "value%20"**

7. **Plus** - `+`

- Matches: **Literal plus character +**
- Purpose: **Used in emails and special formats**
- Example: **"user+tag", "value+add"**

8. **Hyphen/Minus** - `-`

- Matches: **Literal hyphen/minus character -**
- Purpose: **Used in compound words and identifiers**
- Example: **"user-name", "test-value"**

```
*/
```

```
const emailRegex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
console.log(emailRegex.test("user@example.com"));    // true
console.log(emailRegex.test("user.name@co.in"));     // true
console.log(emailRegex.test("user@.com"));           // false
console.log(emailRegex.test("user@com"));            // false


/*6. Password Validation (Strong)
Regex: /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/

Explanation:

(?=.*[a-z]) - At least one lowercase letter

(?=.*[A-Z]) - At least one uppercase letter

(?=.*\d) - At least one digit

(?=.*[@$!%*?&]) - At least one special character

[A-Za-z\d@$!%*?&]{8,} - 8+ characters from allowed set
*/
const strongPassword = /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-
z\d@$!%*?&]{8,}$/;
console.log(strongPassword.test("Password123!"));    // true
console.log(strongPassword.test("weak"));            // false
console.log(strongPassword.test("nouppercase123!")); // false


/*7. Positive Lookahead: (?=...)
The (?=...) syntax is called a positive lookahead assertion. It's a special regex
construct that checks if a pattern exists ahead of the current position, but doesn't
consume any characters.

What It Does:
Checks if the pattern inside (?=...) exists after the current position

Doesn't move the regex pointer forward

Doesn't include the lookahead pattern in the match result

Zero-width - doesn't consume any characters

Syntax: (?=pattern)
*/
const regex = /q(?=u)/; // Match 'q' only if it's followed by 'u'

console.log(regex.test("queen"));  // true - q followed by u
console.log(regex.test("qatar"));  // false - q NOT followed by u
console.log(regex.test("quit"));
```

```
/*8. Username Validation
Regex: /^[a-zA-Z0-9_]{3,20}$/

Explanation:

^[a-zA-Z0-9_] - Start with letter, number, or underscore

{3,20} - 3 to 20 characters long

$ - End of string
*/
const usernameRegex = /^[a-zA-Z0-9._,]{3,20}$/;
console.log(usernameRegex.test("john_doe123"));  // true
console.log(usernameRegex.test("john.doe123"));  // true
console.log(usernameRegex.test("ab"));           // false (too short)
console.log(usernameRegex.test("john@doe"));     // false (special char)


/*9. Date Validation (DD/MM/YYYY)
Regex: /^(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/\d{4}$/

Explanation:

(0[1-9]|[12][0-9]|3[01]) - Day: 01-31
---------------------------------------
0[1-9] - Days 01-09
Explanation:
--------------
0 - Literal zero
[1-9] - Any digit from 1 to 9
Matches: 01, 02, 03, 04, 05, 06, 07, 08, 09

[12][0-9] - Days 10-29
Explanation:
-------------
[12] - Either 1 or 2
[0-9] - Any digit from 0 to 9
Matches: 10, 11, 12, ..., 19, 20, 21, ..., 29

3[01] - Days 30-31
Explanation:
------------
3 - Literal three
[01] - Either 0 or 1
Matches: 30, 31

\/ - Literal slash
```

```
(0[1-9]|1[0-2]) - Month: 01-12
---------------------------------
0[1-9] - Months 01-09
Explanation:
-----------
0 - Literal zero
[1-9] - Any digit from 1 to 9
Matches: 01, 02, 03, 04, 05, 06, 07, 08, 09

1[0-2] - Months 10-12
---------------------
Explanation:
------------
1 - Literal one
[0-2] - Any digit from 0 to 2
Matches: 10, 11, 12

\/ - Literal slash

\d{4} - Year: 4 digits

*/
const dateRegex = /^(0[1-9]|[12][0-9]|3[01])\/(0[1-9]|1[0-2])\/\d{4}$/;
console.log(dateRegex.test("25/12/2023"));   // true
console.log(dateRegex.test("32/12/2023"));   // false (invalid day)
console.log(dateRegex.test("25/13/2023"));   // false (invalid month)


/*10. URL Validation
Regex: /^(https?:\/\/)?([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}(\/\S*)?$/

Explanation:

(https?:\/\/)? - Optional http:// or https://

([a-zA-Z0-9-]+\.)+ - One or more domain parts

[a-zA-Z]{2,} - TLD with 2+ letters

(\/\S*)? - Optional path
*/
const urlRegex = /^(https?:\/\/)?([a-zA-Z0-9-]+\.)+[a-zA-Z]{2,}(\/\S*)?$/;
console.log(urlRegex.test("https://www.example.com"));  // true
console.log(urlRegex.test("example.com/path"));         // true
console.log(urlRegex.test("invalid."));                 // false
```

```
/*11. Credit Card Number (Basic)
Regex: /^\d{4}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}$/

Explanation:

\d{4} - 4 digits

[\s-]? - Optional space or hyphen

Repeated 4 times for 16 digits
*/
const cardRegex = /^\d{4}[\s-]?\d{4}[\s-]?\d{4}[\s-]?\d{4}$/;
console.log(cardRegex.test("1234567812345678"));    // true
console.log(cardRegex.test("1234-5678-1234-5678")); // true
console.log(cardRegex.test("1234 5678 1234 5678")); // true
console.log(cardRegex.test("12345678"));            // false


/*12. IPv4 Address Validation
Regex: /^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-
9][0-9]?)$/

Explanation:

(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?) - Number 0-255

\. - Literal dot

Repeated 3 times, then one final number

*/
const ipRegex = /^((25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(25[0-5]|2[0-4][0-
9]|[01]?[0-9][0-9]?)$/;
console.log(ipRegex.test("192.168.1.1"));    // true
console.log(ipRegex.test("256.168.1.1"));    // false (256 > 255)
console.log(ipRegex.test("192.168.1"));      // false (only 3 parts)

/*13. Time Validation (HH:MM)
Regex: /^([01]?[0-9]|2[0-3]):[0-5][0-9]$/

Explanation:
([01]?[0-9]|2[0-3]) - Hour: 00-23
: - Literal colon
[0-5][0-9] - Minute: 00-59
*/
const timeRegex = /^([01]?[0-9]|2[0-3]):[0-5][0-9]$/;
console.log(timeRegex.test("23:59"));    // true
console.log(timeRegex.test("09:30"));    // true
console.log(timeRegex.test("24:00"));    // false
console.log(timeRegex.test("12:60"));    // false
```