Topic: regex, synchronous and asynchronous

Regular Expressions (Regex)

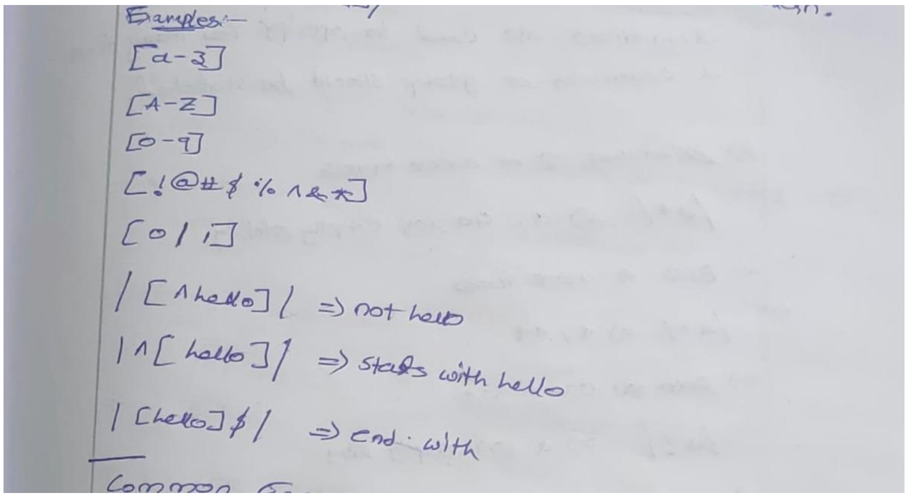
A regular expression is a sequence of characters that helps to create a search pattern, often used for string matching and manipulation.

Ways to Create Regular Expressions

Literal Notation: Uses slashes to define the pattern.

Syntax: /pattern/flags

Example:



Examples:-

[a-z]

[A-Z]

[0-9]

[!@#$%^&*]

[0/1]

/[^hello]/ ⇒ not hello

/^[hello]/ ⇒ starts with hello

/[hello]$/ ⇒ end with

Common

/hello/ matches the string "hello".

/^[0-9]{10}$/ matches a 10-digit number (starts with a digit, ends with a digit).

/[0|1]/ matches either "0" or "1".

Bracket Expressions:

[ ] Matches any character inside the brackets.

Example: [a-z] matches any lowercase letter between 'a' and 'z'.

[^] Matches any character not in the brackets.

Example: [^a-z] matches any character except lowercase letters.

[0-9] Matches any digit between '0' and '9'.

[A-Z] Matches any uppercase letter between 'A' and 'Z'.

Common Escape Sequences

\d Matches any **digit** (equivalent to [0-9]).

\D Matches any **non-digit character** (equivalent to [^0-9]).

\w Matches **alphanumeric characters and underscores** (equivalent to [A-Za-z0-9_]).

\W Matches any **non-alphanumeric character**.

\. Matches a **literal period** (dot).

Methods for Using Regular Expressions

test(): Checks if the pattern exists in a string and returns true or false.

```
const regex = /\d+/;

console.log(regex.test("123"));  // true

console.log(regex.test("abc"));  // false
```

Quantifiers

Quantifiers are used to **specify how many times a character or group should be matched**:

- Matches **zero or more times**.
- **Example:** /a*/ matches "a", "aa", or an empty string.

- + Matches **one or more times**.
- **Example:** /a+/ matches "a", "aa", but not an empty string.

- ? Matches **zero or one time**.
- **Example:** /a?/ matches "a" or an empty string.

- {n}: Matches exactly **n occurrences** of the preceding element.
- **Example:** /a{3}/ matches exactly three "a" characters in a row ("aaa").
- "aaa" matches, "aa" does not match.

- {n,}: **Matches n or more occurrences** of the preceding element.
- **Example:** /a{3,}/ matches three or more "a" characters.
- aaa", "aaaa", and "aaaaa" match, but "aa" does not.

- {n,m}: **Matches between n and m occurrences** of the preceding element.
- **Example:** /a{3,5}/ matches between 3 to 5 "a" characters.
- "aaa", "aaaa", and "aaaaa" match, but "aa" and "aaaaaa" do not.

Examples of Regex

Phone Number Validation:
Regex: /^[0-9]{10}$/
Explanation: Matches exactly 10 digits from 0-9, used for validating phone numbers.

Binary Values (0 or 1):
Regex: /[0|1]/
Explanation: Matches either 0 or 1.

Simple Email Validation:
Regex: /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
Explanation: Matches a basic email format.

Examples

1. Validate an Email Address
Pattern: /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/
Explanation:
^[a-zA-Z0-9._%+-]+ - Matches the username (alphanumeric characters and special symbols like ._%+-).
@ - Requires the "@" symbol.
[a-zA-Z0-9.-]+ - Matches the domain name.
\.[a-zA-Z]{2,} - Ensures a valid top-level domain (TLD), like .com or .net.

2. Validate a Phone Number (10 digits)
Pattern: /^[0-9]{10}$/
Explanation: ^[0-9]{10}$ - Ensures exactly 10 digits between 0 and 9.

3. Validate a URL
Pattern: /^(https?:\/\/)?(www\.)?[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}(\/\S*)?$/
Explanation:
https?:\/\/ - Matches "http" or "https".
(www\.)? - Optional "www.".
[a-zA-Z0-9.-]+ - Matches the domain name.
\.[a-zA-Z]{2,} - Ensures a valid TLD.
(\/\S*)? - Optionally allows a path after the domain.

4. Validate a Credit Card Number
Pattern: /^(?:\d{4}-?){3}\d{4}$/
Explanation:
(?:\d{4}-?){3} - Matches three groups of 4 digits, optionally separated by hyphens.
\d{4} - Matches the final group of 4 digits.

5. Validate a ZIP Code (US, 5 digits)
Pattern: /^\d{5}(-\d{4})?$/
Explanation:
^\d{5} - Matches exactly 5 digits.
(-\d{4})? - Optionally matches a hyphen followed by 4 digits for ZIP+4 codes.

6. Match a Date (MM/DD/YYYY)
- **Pattern:** /^(0[1-9]|1[0-2])\/(0[1-9]|[12][0-9]|3[01])\/\d{4}$/
- **Explanation:**
- (0[1-9]|1[0-2]) - Matches months from 01 to 12.
- (0[1-9]|[12][0-9]|3[01]) - Matches days from 01 to 31.
- \d{4} - Matches exactly 4 digits for the year.

7. Match a Time (24-hour format)
- **Pattern**: /^([01][0-9]|2[0-3]):[0-5][0-9]$/
- **Explanation**:
- ([01][0-9]|2[0-3]) - Matches hours from 00 to 23.
- :[0-5][0-9] - Matches minutes from 00 to 59.

8. Match Only Numbers
Pattern: /^\d+$/
Explanation:
^\d+$ - Matches any sequence of one or more digits (whole numbers only).

9. Match Only Alphanumeric Characters
Pattern: /^[a-zA-Z0-9]+$/
Explanation: ^[a-zA-Z0-9]+$ - Matches any sequence of alphanumeric characters.

10. Match Hexadecimal Colors
Pattern: /^#?([a-fA-F0-9]{6}|[a-fA-F0-9]{3})$/
Explanation:
#? - The # symbol is optional.
([a-fA-F0-9]{6}|[a-fA-F0-9]{3}) - Matches either a 6-character or 3-character hex color.

11. Strip Whitespace from the Beginning and End of a String
Pattern: /^\s+|\s+$/g
Explanation:
^\s+ - Matches one or more whitespace characters at the start.
\s+$ - Matches one or more whitespace characters at the end.
g flag - Global search for all matches.

12. Match a Word Boundary
Pattern: /\bword\b/
Explanation:
\b - Ensures the pattern matches at word boundaries.
word - The specific word to match.

13. Match a Floating-Point Number
Pattern: /^[+-]?([0-9]*[.])?[0-9]+$/
Explanation:
[+-]? - Allows an optional "+" or "-" sign.
([0-9]*[.])? - Optionally matches digits before and after a decimal point.
[0-9]+ - Requires at least one digit.

14. Validate a Strong Password
Pattern: /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/
Explanation:
(?=.*[a-z]) - Requires at least one lowercase letter.
(?=.*[A-Z]) - Requires at least one uppercase letter.
(?=.*\d) - Requires at least one digit.
(?=.*[@$!%*?&]) - Requires at least one special character.
{8,} - Must be at least 8 characters long.

15. Match a Repeated Character
Pattern: /(\w)\1+/
Explanation:
(\w) - Captures any alphanumeric character.

\1+ - Matches one or more occurrences of the same captured character.

16. Match an IP Address (IPv4)
Pattern: /^(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/
Explanation:
Each part of the IP is matched using (25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?) which ensures a valid number between 0 and 255.

17. Match HTML Tags
Pattern: /^<\/?[\w\s"'-=]+>$/
Explanation:
<\/? - Matches an opening or closing tag.
[\w\s"'-=]+ - Matches tag content (attributes, values).
> - Matches the closing bracket.

18. Match Leading Zeroes
Pattern: /^0+(?=\d)/
Explanation:
^0+ - Matches one or more leading zeroes.
(?=\d) - Ensures a digit follows.

19. Validate a Username
Pattern: /^[a-zA-Z0-9_]{3,16}$/
Explanation:
^[a-zA-Z0-9_]{3,16}$ - Matches a username that is 3-16 characters long, only allowing alphanumeric characters and underscores.

20. Match a Word with Only Letters
Pattern: /^[A-Za-z]+$/
Explanation:
^[A-Za-z]+$ - Matches a word containing only letters (upper or lowercase).

Synchronous and asynchronous

Synchronous: In synchronous operations, code is executed sequentially, one line at a time. Each line must wait for the previous one to finish executing before it can start. This can sometimes lead to blocking behavior, where one task prevents another from executing until it's complete.

```
console.log("Start");
console.log("Middle");
console.log("End");
// start
// middle
// end
```
In this synchronous code, "Start" will be logged first, followed by "Middle", and then "End".

Asynchronous: Asynchronous operations allow code to execute independently from the main program flow. This means that while one operation is being processed, the program can continue to execute other tasks. Asynchronous operations are typically used for tasks that may take some time to complete, such as fetching data from a server or reading a file. In JavaScript, common asynchronous operations are handled using callbacks, promises, or async/await syntax.

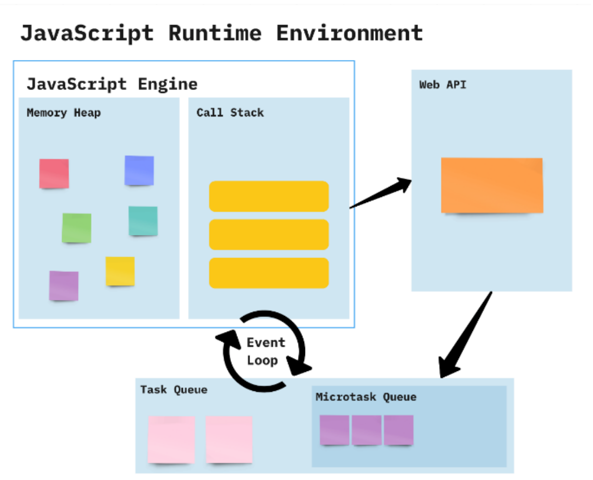
```
console.log("Start");

setTimeout(()=>{
  console.log("middle")
}),2000;
console.log("End");

// start
// end
// middle
```

In this example, "Start" is logged first, then after a delay of 2000 milliseconds, "End" is logged, followed by " Middle ".

How js works



JavaScript works on a environment called **JavaScript Runtime Environment**. To use JavaScript you basically install this environment and than you can simply use JavaScript.

So in order to use JavaScript you install a **Browser** or **NodeJS** both of which are JavaScript Runtime Environment.

call stack:

The call stack is used to store information about function calls, including local variables, parameters, and the point of execution.

Heap:

The heap is a region of memory used for dynamic memory allocation. It stores objects, arrays, and other complex data structures that are created and managed at runtime.

Web API:

A Web API (Application Programming Interface) is a set of rules and tools that allows different software applications to communicate with each other over the web. It acts as an intermediary, enabling one application to interact with another application's data or functionality using standard web protocols, usually HTTP.

Event loop:

The event loop is a fundamental concept in asynchronous programming, especially in environments like JavaScript. It enables non-blocking operations, allowing code to execute asynchronously while ensuring that tasks are handled in an orderly manner. Here's a closer look at how synchronous and asynchronous operations interact with the event loop:

Synchronous vs. Asynchronous

Synchronous:

Synchronous operations are executed sequentially, one after the other. Each operation must complete before the next one begins.

Asynchronous:

Asynchronous operations allow code to execute without waiting for previous operations to complete. This is useful for tasks that involve waiting, such as network requests or timers.

CallStack Overflow: If the call stack grows too large, typically due to infinite recursion or excessive nested function calls, it can exceed the available memory allocated for the stack. This results in a "stack overflow" error and crashes the program.

How it Works:

- Constantly checks whether or not the call stack is empty
- When the call stack is empty, all queued up Microtasks from Microtask Queue are popped onto the callstack
- If both the call stack and Microtask Queue are empty, the event loop dequeues tasks from the Task Queue and calls them
- Starved event loop

1. Basic Synchronous Code

```
console.log("Task 1: Start");
console.log("Task 2: Middle");
console.log("Task 3: End");

// start
// middle
// end
```

2 Asynchronous with setTimeout (Event Loop in Action)

```
console.log("Task 1: Start");

setTimeout(() => {
  console.log("Task 2: Asynchronous Task (After 2 seconds)");
}, 2000);

console.log("Task 3: End");
```

```
start
// end
// asynchronous task
```

3. Asynchronous Code with a Promise

```
console.log("Task 1: Start");

const promise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Task 2: Promise Resolved (After 1 second)");
  }, 1000);
});

promise.then((message) => {
  console.log(message);
});

console.log("Task 3: End");

// Task 1: Start
// Task 3: End
// Task 2: Promise Resolved (After 1 second)
```

4. Multiple Asynchronous Tasks

```
 console.log("Task 1: Start");

  setTimeout(() => {
    console.log("Task 2: Asynchronous Task 1 (After 2 seconds)");
  }, 2000);

  setTimeout(() => {
    console.log("Task 3: Asynchronous Task 2 (After 0 seconds)");
  }, 0);

  console.log("Task 4: End");

  // Task 1: Start
  // Task 4: End
  // Task 3: Asynchronous Task 2 (After 0 seconds)
  // Task 2: Asynchronous Task 1 (After 2 seconds)
```

5. setTimeout and Promise Together

```
console.log("Task 1: Start");

setTimeout(() => {
  console.log("Task 2: Asynchronous Task (setTimeout)");
}, 0);

Promise.resolve().then(() => {
    console.log("Task 3: Promise Resolved");
});

console.log("Task 4: End");

// Task 1: Start
// Task 4: End
// Task 3: Promise Resolved
// Task 2: Asynchronous Task (setTimeout)
```

6. Using async/await

async/await is a syntactic sugar for working with promises. It behaves synchronously within an async function until an await keyword is encountered, at which point it returns to the event loop to handle other tasks.

```
console.log("Task 1: Start");

    async function asyncTask() {
        console.log("Task 2: Inside asyncTask");
        await new Promise(resolve => setTimeout(resolve, 1000)); // Wait for 1 second
        console.log("Task 3: After 1 second wait in asyncTask");
    }

    asyncTask();

    console.log("Task 4: End");


//   Task 1: Start
//    Task 2: Inside asyncTask
//    Task 4: End
//    Task 3: After 1 second wait in asyncTask
```

7) setTimeout with Different Delays

```
   console.log("Task 1: Start");

    setTimeout(() => {
        console.log("Task 2: 2 seconds delay");
    }, 2000);

    setTimeout(() => {
        console.log("Task 3: 1 second delay");
    }, 1000);

    setTimeout(() => {
        console.log("Task 4: 0 seconds delay");
    }, 0);

    console.log("Task 5: End");

    // Task 1: Start
    // Task 5: End
    // Task 4: 0 seconds delay
    // Task 3: 1 second delay
    // Task 2: 2 seconds delay
```