

# FUTURA

# LA SCUOLA PER L'ITALIA DI DOMANI



Finanziato  
dall'Unione europea  
NextGenerationEU



Ministero dell'Istruzione  
e del Merito



**Italiadomani**  
PIANO NAZIONALE DI RIPRESA E RESILIENZA

DOCENTE	Shadi Lahham
Corso	Web Developer
Unità Formativa	Programmazione - Javascript e Typescript
Argomento	Specificato nel titolo della slide successiva

Intervento realizzato da  
**ITS**  
TECNOLOGIE  
DELL'INFORMAZIONE  
E DELLA COMUNICAZIONE

COESIONE  
ITALIA 21-27  
PIEMONTE



Cofinanziato  
dall'Unione europea



REGIONE  
PIEMONTE

# Regular expressions

Power Patterns

Shadi Lahham - Web development

# Regex patterns

# Pattern: /a/gm

ag

gaga

alpha

omega

banana

car

AG

gammA

delta

apple

Abacus

planetary

## Specific character matching

The pattern `/a/gm` matches any occurrence of the letter "a" in a string, globally and across multiple lines

# Pattern: /ar/gm

articulate park

caring farmer

farmland

parrot in the park

marketplace

zoom out

barricade the road

ARMY barracks

cartoons on TV

guitarist in the band

## Specific character sequence matching

The pattern `/ar/gm` matches any occurrence of the letters "ar" in a string, globally and across multiple lines

# The global /g flag

Pattern `/ar/m`

the articulate park

caring farmer

farmland

parrot in the park

marketplace

zoom out

barricade the road

ARMY barracks

cartoons on TV

guitarist in the band

The global flag

The `/g` flag in regular expressions stands for "global" and it's used to perform a global search in a string meaning that instead of stopping after the first match, the search continues to find all matches in the entire string

# The insensitive /i flag

Pattern /ar/gim

articulate pARk

caring faRmer

farmland

parrot in the pArk

marketplace

zoom out

barricade the road

ARMY barracks

cartoons on TV

guitarist in the band

The insensitive flag

The /i flag in regular expressions stands for "case-insensitive matching". It's used to perform matches without distinguishing between uppercase and lowercase letters.

# Pattern: /a|b/gm

ab

cba

alphabet

bobcat

amazing

ba\_lloon

robin

batman

cabbage

bubblegum

carbon

crabgrass

## Alternation operator

The pattern `/a|b/gm` matches either the letter "a" or the letter "b" globally and across multiple lines within a string



# Pattern: /a|br/gm

abc

bravo

abacus

bristle

alphabet

cobra

cabaret

breath

abrasive

cabbbage

arbitraate

a brisk waalk

## Alternation operator

The pattern `/a|br/gm` matches either the letter "a" or the sequence "br" globally and across multiple lines within a string

# Pattern: /employ(er|ee|ment|ing|able)/gm

unemployment  
employer  
happy employees  
employing  
unemployable  
unemployed  
deployment  
employingment  
reemploy  
employmentable  
disemploying  
employinging

## Alternation operator with parentheses

The pattern /employ(er|ee|ment|ing|able)/gm matches any string containing the word "employ" followed by any of the specified suffixes: "er", "ee", "ment", "ing", or "able"

## Additional example

```
/the (red|green|blue) pixel/gm  
the green pixel  
the red pixel  
the blue pixel
```

# Parentheses

In regular expressions, parentheses `()` serve two main purposes

1. **Grouping:**  
Parentheses are used to group together multiple characters or subpatterns. This grouping allows applying quantifiers (such as `*`, `+`, `?`, or `{}`) to the entire group, treating it as a single unit
2. **Capturing:**  
Parentheses are also used to create capture groups. When a pattern is matched, the content matched by the expression inside the parentheses is captured and can be referenced or extracted later in the regex or in code using backreferences

# Pattern: /s[aio]r/gm

spark

yes sir

be sour

less sore

sorcerer

soar

sira sing a sor

sugar

sirasar

saior

my savior

to scatter

## Character set

The pattern `/s[aio]r/gm` matches any word containing the letter "s" followed by either "a", "i", or "o", and ending with "r", globally and across multiple lines within a string

# Pattern: /s[a-g]r/gm

hapserstar  
pesgrybr  
a sirlayer  
sarpusspur  
extra esbrugar  
sagr  
sinsgri  
ent esari ba  
comsbricum  
scrape  
asdro  
exeser

## Character range

The pattern /s[a-g]r/gm matches strings that start with 's', followed by any letter from 'a' to 'g', and end with 'r', globally and across multiple lines within a string

# Pattern: /s[^abc]t/gm

seat with spt-ray

set without yam

sot within zoo

sort beyond sitce

site beside 123

adjacent sut to desk

sct sbt sat set asit espt

## Negated character set

The pattern /s[^abc]t/gm matches any occurrence of a word that starts with "s", followed by any character except "a", "b", or "c", and ends with "t", globally and across multiple lines

# Pattern: /p[a-zA-Z]a/gm

appramant

apxa

sopea

a piazza

to prada

paaaaapoal-ippa

psa public

pharma

enterprand

mpAai

abpXa

abPXa

The pattern `/p[a-zA-Z]a/gm` matches any occurrence of a three-letter word where the first and third letters are "p" and "a" respectively, and the second letter can be any alphabet character either lowercase or uppercase, globally and across multiple lines within a string

## Common patterns

The pattern `/[A-Za-z0-9]/gm` matches just one alphanumeric character in a string

## The meta sequence `\w`

Matches any letter, digit or underscore and is equivalent to `[a-zA-Z0-9_]`

# Meta sequences

Meta sequences in regex are shorthand characters representing commonly used classes

`\d` Matches any digit character (equivalent to `[0-9]`)

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

`\w` Matches any word character (alphanumeric character plus underscore) (equivalent to `[A-Za-z0-9_]`)

`\W` Matches any non-word character (equivalent to `[^A-Za-z0-9_]`)

`\s` Matches any whitespace character (space, tab, newline, [CRLE](#), etc)

`\S` Matches any non-whitespace character



# Pattern: /\d\d-\w\w\w\s\w\d/gm

12-abc X5

9a-xyz A1

34-p!r B2

87-mno C3

55-uvw\_D4

78-ghi \_5t34

23-lmn F6-fa-23-ujn u2

45-def-G7

67-rst.H8

88-jkl0I9

10-222 20

32-abc K1

## Meta sequences

The pattern /\d\d-\w\w\w\s\w\d/gm matches strings that follow a specific format:

\d\d: Two digits

-: A hyphen

\w\w\w: Three word characters

\s: A whitespace character

\w: A word character

\d: A digit

So, the pattern matches strings that start with two digits followed by a hyphen, then three word characters, a whitespace character, a single word character, and finally a digit

# Pattern: /a.b.c.d/gm

a1b2c3d

avbyczdmn

aAbBcCd

a!b@c#d7a7bec9d9d

a.b.c.d

a12bb456c789d

aabbcc^d

aabbccd

a.b#cd

a2b4c6d8

abcdef

a,b.c?d

The pattern `/a.b.c.d/gm` matches any string that contains the sequence "a", followed by any character, then "b", followed by any character, then "c", followed by any character, and finally "d", globally and across multiple lines within a string

## Any single character

The dot (.) in a regular expression matches any single character except newline

# Start of string & end of string

In regular expressions, the caret `^` is used to denote the start of a string, and the dollar sign `$` is used to denote the end of a string

`^`: Matches the start of a string

`$`: Matches the end of a string

When using `^`, the pattern that follows must appear at the beginning of the string

Similarly, when using `$`, the pattern that precedes it must appear at the end of the string

# Start of string

Pattern `/^send/gm`

sending the request

don't send

sender's name

wesend

The caret `^`

`^`: Matches the start of a string

When using `^`, the pattern that follows must appear at the beginning of the string

# Start of string & end of string

Pattern `/send$/gm`

sending the request

don't send

sender's name

wesend

The dollar sign `$`

`$`: Matches the end of a string

When using `$`, the pattern that precedes it must appear at the end of the string

# The multiline /m flag

Pattern `/^send/g`

sending the request

don't send

sender's name

wesend

Pattern `/send$/g`

sending the request

don't send

sender's name

wesend

## The multiline flag

The `/m` flag in regular expressions stands for "multiline" and it's used to modify the behavior of the `^` and `$` anchors

With this flag, `^` matches the start of a line and `$` matches the end of a line, in addition to matching the start and end of the entire string

Without this flag, `^` matches only the start of the entire string and `$` matches only the end of the entire string, disregarding line breaks

# Quantifiers

Quantifiers specify repetition in regex for character, group, or class matching, enhancing flexibility and power

## Basic quantifiers

- \* (Zero or more): Matches zero or more occurrences of the preceding element
- + (One or more): Matches one or more occurrences of the preceding element
- ? (Zero or one): Matches zero or one occurrence of the preceding element

## Explicit quantifiers

- {n} (Exactly n): Matches exactly n occurrences of the preceding element
- {n,} (At least n): Matches at least n occurrences of the preceding element
- {n,m} (Between n and m): Matches between n and m occurrences of the preceding element

## Lazy quantifiers

The previous quantifiers are 'greedy'

Adding a ? makes them 'lazy' meaning they match as few characters as possible

\*?, +?, ??, {n}?, {n,}?, {n,m}?

# Pattern: /a\*b/gm

a123b

axb

ab

b

cb

aaaaab

aab

cabcabcab

bacbacb

mbbba

## Zero or more

The asterisk (\*) in regular expressions is called the "asterisk" or "star" quantifier

It specifies that the preceding character or group can occur zero or more times

/a\*/ matches "", "a", "aa", "aaa", etc

/ab\*/ matches "a", "ab", "abb", "abbb", etc



# Pattern: /a?b/gm

a123b

axb

ab

b

cb

aaaaab

aab

cabcabcab

bacbacb

mbbba

## Zero or one

The question mark (?) in regular expressions is called the "question mark" or "zero or one" quantifier

It specifies that the preceding character or group can occur zero or one time

/colou?r/ matches "color" and "colour" but not "colouur"

/ab?c/ matches "ac" and "abc" but not "abbc"

# Pattern: /a+b/gm

a123b

axb

ab

b

cb

aaaaab

aab

cabcabcab

bacbacb

mbbba

## One or more

The plus sign (+) in regular expressions is called the "plus" quantifier

It specifies that the preceding character or group must occur one or more times

/a+/ matches "a", "aa", "aaa", etc but not ""

/ab+/ matches "ab", "abb", "abbb", etc not "a"

# Pattern: /a{3}b/gm

aaacb

ab

aab

aaab

bxaaabcvaaab

aaaaab

## Exactly n

The curly braces {} with a specific number (n) inside in regular expressions is used to denote the "exact quantity" quantifier

It specifies that the preceding character or group must occur exactly n times

/a{3}/ matches "aaa"

/b{2}/ matches "bb", but not "b" or "bbb"

# Pattern: /a{3,}b/gm

aaacb

ab

aab

aaab

bxaaabcvaaab

aaaaab

baaabxaaaaaaaaab

## At least n

The curly braces {} with a specific number (n) followed by a comma (,) but without a maximum value (m) inside in regular expressions is used to denote the "at least n" quantifier

It specifies that the preceding character or group must occur at least n times

/a{2,}/ matches "aa", "aaa", "aaaa", etc not "a"  
/b{3,}/ matches "bbb", "bbbb", etc but not "bb"  
or "b"

# Pattern: /a{2,4}b/gm

aaacb

ab

aab

aaab

bxaaabcvaaab

aaaaab

baaabxaaaaaaaab

## Between n and m

The curly braces {} with a specific range of numbers (n and m) inside, separated by a comma (,) in regular expressions is used to denote the "between n and m" quantifier

It specifies that the preceding character or group must occur between n and m times, inclusive

/a{2,4}/ matches "aa", "aaa", or "aaaa" but not "a" or "aaaaa"

/b{1,3}/ matches "b", "bb", or "bbb" but not "bbbb" or an empty string

# Escape character \

The backslash (\) is used as the escape character to indicate that the character immediately following it should be treated as a literal character

`/a\*/` matches "a\*" instead of treating "\*" as a quantifier

`/\\d/` matches "\d" instead of interpreting "\d" as a digit character

`/b\.com/` matches "b.com" instead of treating "." as a wildcard for any character

## Commonly escaped characters

`\.` (period)

`\*` (asterisk)

`\+` (plus)

`\?` (question mark)

`\{` (left curly brace)

`\}` (right curly brace)

`\[` (left square bracket)

`\]` (right square bracket)

`\\` (backslash)

# Regex with Javascript

# Creating a RegEx

*// using new RegExp*

```
const pattern = new RegExp('hello', 'i'); // new RegExp(pattern, flags)
const text = 'Hello, world!';
console.log(pattern.test(text)); // true
```

*// using literal regular expression*

```
const pattern = /hello/i; // literal notation
const text = 'Hello, world!';
console.log(pattern.test(text)); // true
```



# String methods using RegEx

*// match() returns an array containing all matches of a pattern in a string.*

```
const text = 'Hello, world!';  
const matches = text.match(/[a-z]+/gi); // matches all words  
console.log(matches); // ["Hello", "world"]
```

*// replace() replaces matches of a pattern with a specified replacement string.*

```
const text = 'Hello, world!';  
const newText = text.replace(/[aeiou]/gi, '*'); // replaces vowels with '*'  
console.log(newText); // 'H*LL*, w*rld!'
```

*// search() returns the index of the first match of a pattern in a string.*

```
const text = 'Hello, world!';  
const index = text.search(/world/i); // searches for 'world'  
console.log(index); // 7
```

# Regex methods

*// test() tests for a match in a string. Returns true or false.*

```
const pattern = /[0-9]+/; // literal notation
```

```
console.log(pattern.test('abc123')); // true
```

*// exec() executes a search for a match in a specified string*

*// returns an array of information or null if no match is found*

```
const pattern = /[0-9]+/;
```

```
console.log(pattern.exec('abc123')); // ["123"]
```

Your turn

# 1.Regex validation

Write regular expressions to validate the following inputs

## 1.Email Address

Expected pattern: `[any characters]@[any characters].[2-4 letters]`

## 2.Phone Number

Expected pattern: `[optional + or country code] [digits, possibly separated by dashes or spaces]`

## 3.Password

Expected pattern: `[at least 8 characters, including at least one uppercase letter, one lowercase letter, one digit, and one special character]`

## 4.URL

Expected pattern: `[protocol]://[domain].[top-level domain]/[optional path]?[optional query string]#[optional fragment]`

## Note

Invent multiple test cases to thoroughly test your regular expressions

# References

[Regular Expressions Guide: Everything You Need to Know](#)  
[JavaScript RegExp Object](#)

MDN

[Regular expressions MDN](#)

[RegExp | MDN](#)

[RegExp\(\) constructor | MDN](#)

[Regular expression syntax cheat sheet | MDN](#)

# References

Tools

[Regex101](#)

[RegExr](#)

[Regex Generator](#)

# © Copyright & Attribution

Unless otherwise stated, all materials are © 2017–2025 [Shadi Lahham](#)

For personal use only. May not be shared or reproduced without written permission

Brief excerpts may be used with proper attribution

External links and resources are copyrighted by their respective owners