AIMERZ.ai
Aim.Act.Achieve

NOTES

Regex in javascript

Regex in Javascript

Introduction to regex

A regular expression (regex) is a pattern used to match character combinations in strings. It allows searching, replacing, and validating text using special syntax for specific characters (e.g., \d for digits). Commonly used for tasks like validation, search, and extraction in text processing.

Declaration:

In general there are two ways to declare a regular expression:-

1. **Using Constructor**

The Constructor RegEx(“<Expression>”) is used to declare a regular expression in javascript.

Example:

```javascript
var regexConst = new RegExp('abc');;
console.log(regexConst);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
/abc/
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

2. **Using RegEx Literal**

This Literal is another way to declare a regular expression. Regular Expression Literal is used when we want to create regular expressions dynamically, in which case, regex literal won't work.

Example:

```javascript
var regexLiteral = /def/;
console.log(regexLiteral);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
/def/
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

RegEx Object & String Methods

A Regular expression object is a kind of bundle of properties which are pre-defined in the form of objects and methods.

There are various methods to use the RegExp object to test, match, and validate various string values in JavaScript :-

1. **test()**
2. **match()**
3. **replace()**
4. **exec()**

Now lets discuss each of them one by one :-

1. **test() :** The test() is a method on a regular expression object that tests if a pattern exists within a string. It returns true if there's a match and false otherwise.

   Example:

```javascript
var pattern = /hello/;
```

```
var result = pattern.test("hello world");  // Returns true

console.log(result);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

2. **match():** The match() method is used on strings to search for matches with a regular expression. It returns an array of matches or null if no match is found.

   Example:

```
JavaScript
const text = "hello world";
const result = text.match(/world/);

console.log(result);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
[ 'world', index: 6, input: 'hello world', groups: undefined ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

Explanation:

- 'world': This is the matched substring.
- index: 6: This indicates the starting position of the match within text.
- input: 'hello world': This is the original string.
- groups: undefined: If there were capturing groups in the pattern, they would appear here.

3. **replace():**The replace() method on strings replaces occurrences of a specified substring or pattern with another string. It can accept a regular expression as the pattern to replace.

Example:

```javascript
const text = "hello world";
const newText = text.replace(/world/, "JavaScript");


console.log(result);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
hello JavaScript
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

4. **exec():** The exec() method is used on a regular expression object to execute a search for a match in a specified string. It returns an array of matched results or null if no match is found, and provides more detailed match information than match().

Example:

```javascript
const pattern = /world/;
const result = pattern.exec("hello world");

console.log(result);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
[ 'world', index: 6, input: 'hello world', groups: undefined ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

Characters in regex

There are many characters in regular expressions as listed here:-

1. **+ (Plus)**
2. *** (asterisk)**
3. **? (Question Mark)**
4. **^ (Caret)**
5. **{ } (Curly Braces)**
6. **[ ] (Square Brackets)**

7. **| (Pipe)**
8. **( ) (Parentheses)**

Now lets discuss all of them one by one :-

1. **+ (Plus) :** Matches one or more occurrences of the preceding character.

   Example:

```javascript
let regexPlus = /a+/;
console.log(regexPlus.test("aaab"));
// Output: true (matches one or more 'a')
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

2. *** (asterisk) :** It matches zero or more occurrences of the preceding character.

Example:

```javascript
let regexAsterisk = /a*/;
console.log(regexAsterisk.test("b"));

// Output: true (matches zero 'a')
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

3. **? (Question Mark):** Matches zero or one occurrence of the preceding character.

Example:

```javascript
let regexQuestion = /colou?r/;
console.log(regexQuestion.test("color"));
// Output: true (matches "color")

console.log(regexQuestion.test("colour"));
 // Output: true (matches "colour")

console.log(regexQuestion.test("colr"));
 // Output: false (does not match "colr")
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
true
false
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

4. **^ (Caret):** It matches the beginning of the string, the regular expression that follows it should be at the start of the test string, i.e the caret (^) matches the start of the string.

Example:

```javascript
let regexCaret = /^hello/;
console.log(regexCaret.test("hello world"));
 // Output: true (matches at the beginning)
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

5. **$ (Dollar) :** Matches the end of the string. That is, the regular expression that precedes it should be at the end of the test string. The dollar ($) sign matches the end of the string.

Example:

```javascript
let regexDollar = /world$/;
console.log(regexDollar.test("hello world"));
// Output: true (matches "world" at the end of the string)

console.log(regexDollar.test("world hello"));
// Output: false (does not match "world" at the end)

console.log(regexDollar.test("hello world!"));
// Output: false (does not match "world" without exact end)
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
true
false
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

6. **{ } (Curly Braces) :** Specifies the exact number of repetitions for the preceding element.

Example:

```javascript
let regexCurly = /a{2,4}/;
console.log(regexCurly.test("aaab"));
// Output: true (matches 'aaa' which is between 2 and 4 'a's)
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

7. **[ ] (Square Brackets) :** The square brackets [ ] define a character class, matching any one character within them.

   For example: [abc] matches "a," "b," or "c".

Example:

```javascript
let regexSquare = /[aeiou]/;
console.log(regexSquare.test("apple"));
 // Output: true (matches one of the vowels)
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

8. **| (Pipe) :** In regular expressions, the pipe symbol (|) acts as an "OR" operator, matching either the expression on its left or the one on its right.

Example:

```javascript
let regexPipe = /cat|dog/;
console.log(regexPipe.test("I have a dog"));
// Output: true (matches "dog")

console.log(regexPipe.test("I have a bird"));
// Output: false (does not match "cat" or "dog")
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
false
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

9. **( ) (Parentheses) :** The ( ) (parentheses) group characters or expressions, treating them as a single unit. They capture matched text for later use or reference.

Example:

```javascript
let regexParentheses = /(hello) (world)/;
console.log(regexParentheses.test("hello world"));
 // Output: true (matches "hello world")

console.log(regexParentheses.test("hello everyone"));  // Output: false (does not match "world")
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST> node logic.js
true
false
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\TEST>
```

Special Character Classes

1. **\d:** It matches any digit (0-9) with the provided Regular Expression.

   Example:

```javascript
const text = "Hello World! 123_ ABC\tNewline\n";

// \d: Matches any digit (0-9)
const digits = text.match(/\d/g);
console.log("\\d (digits):", digits);
// Output: ["1", "2", "3"]
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\d (digits): [ '1', '2', '3' ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

2. **\D:** Matches any **non-digit** with the provided Regular Expression.

Example:

```
JavaScript
const text = "Hello World! 123_ ABC\tNewline\n";

const nonDigits = text.match(/\D/g);
console.log("\\D (non-digits):", nonDigits);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\D (non-digits): [
  'H',  'e', 'l',  'l', 'o', ' ',
  'W',  'o', 'r',  'l', 'd', '!',
  ' ',  '_', ' ',  'A', 'B', 'C',
  '\t', 'N', 'e',  'w', 'l', 'i',
  'n',  'e', '\n'
]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

3. **\w:** Matches any word character **(alphanumeric + underscore).**

Example:

```
JavaScript
const text = "Hello World! 123_ ABC\tNewline\n";

const wordCharacters = text.match(/\w/g); console.log("\\w (word characters):",
wordCharacters);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\w (word characters): [
  'H', 'e', 'l', 'l', 'o',
  'W', 'o', 'r', 'l', 'd',
  '1', '2', '3', '_', 'A',
  'B', 'C', 'N', 'e', 'w',
  'l', 'i', 'n', 'e'
]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

4. **\W:** This matches any **non-word** character with the provided regular expression.

Example:

```javascript
const text = "Hello World! 123_ ABC\tNewline\n";

const nonWordCharacters = text.match(/\W/g);
console.log("\\W (non-word characters):", nonWordCharacters);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\W (non-word characters): [ ' ', '!', ' ', ' ', '\t', '\n' ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

5. **\s:** It matches any **whitespace (space, tab, newline)** with the regular expression.

Example:

```javascript
const text = "Hello World! 123_ ABC\tNewline\n";

const whitespaces = text.match(/\s/g);
console.log("\\s (whitespace):", whitespaces);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\s (whitespace): [ ' ', ' ', ' ', '\t', '\n' ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

6. **\S:** It matches any **non-whitespace with the regular expression** provided.

Example:

```javascript
const text = "Hello World! 123_ ABC\tNewline\n";

const nonWhitespaces = text.match(/\S/g);
console.log("\\S (non-whitespace):", nonWhitespaces);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test> node logic.js
\S (non-whitespace): [
  'H', 'e', 'l', 'l', 'o', 'W',
  'o', 'r', 'l', 'd', '!', '1',
  '2', '3', '_', 'A', 'B', 'C',
  'N', 'e', 'w', 'l', 'i', 'n',
  'e'
]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\test>
```

Regex with example

Example: Write a regular expression to check the correct format of an email address by using Javascript.

```javascript
// Sample email to test
let email = "example.email123@example-domain.com";

// Regular expression pattern for an email address
let pattern = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;

// Check if the email matches the pattern
if (pattern.test(email)) {
    console.log("Valid email format.");
} else {
    console.log("Invalid email format.");
}
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
Valid email format.
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

String

Introduction to string

Strings are useful for holding data that can be represented in text form. In other words strings are objects that represent a sequence of characters. They come with a variety of built-in methods and properties to handle text manipulation.

Declaration of String

1. **Using String Primitives**
2. **Using String Constructor**

1. Using String Primitives:

A string primitive is a sequence of characters represented as a primitive data type rather than an object.

Syntax:

```
JavaScript
const string1 = "A string primitive";
```

2. Using String Constructor:

We can create strings using the String constructor, though it's typically more efficient to use string literals. The String constructor creates a String object rather than a string primitive.

Syntax:

```javascript
const string4 = new String("A String object");
```

Using different quotes for String

In JavaScript, you can define strings using three types of quotes: single quotes ('...'), double quotes ("..."), and backticks (`...`). Each type has its own advantages and specific use cases.

Let us see the above with the help of an example :-

1. Single Quotes ('...'):

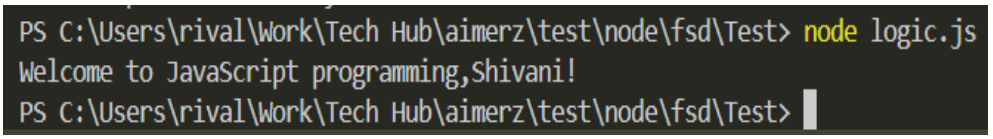
Single quotes are commonly used for simple string literals. They work well for strings that don't need special formatting or interpolation.

Example:

```javascript
let name = 'Shivani';
let message ='Welcome to JavaScript programming,'+ name + '!';

console.log(message);
// Output: Welcome to JavaScript programming, Shivani!
```

OUTPUT:



2. Double Quotes ("..."):

Double quotes are similar to single quotes but are useful when you have single quotes within the string. This helps avoid the need for escaping single quotes.

Example:

```javascript
let sentence = "JavaScript's flexibility is remarkable!";

console.log(sentence);
// Output: JavaScript's flexibility is remarkable!
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
JavaScript's flexibility is remarkable!
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

3. Backticks (Template Literals) (`...`):

Backticks (also known as template literals) allow us to make multi-line strings. We do not need to use \n for new lines, also we can use string interpolation whenever required.

Note: String interpolation is the process to embed expressions with ${}.

Example:

```javascript
let name = "Shivani";
let message = `Hello, ${name}!
Welcome to the world of JavaScript.`;
console.log(message);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
Hello, Shivani!
Welcome to the world of JavaScript.
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Property Access & its Problems

We can access specific characters in a string using property access (indexing), which returns the character at a particular position.

Example: Create a string and print the characters at position 3,4 & 6 indexes.

```javascript
let str = "JavaScript";

console.log(str[3]+ " " + str[4] + " " + str[6]);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
a S r
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Problems with Property Access

1. Immutability of Strings

Strings are immutable, meaning any attempt to modify a character within a string won't work. Even if you try using bracket notation to change a character, the string remains unchanged.

Example:

```javascript
let text = "Hello";
text[0] = "Y"; // Attempt to change "H" to "Y"
console.log(text); // Output: "Hello" (no change)
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
Hello
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

2. Inefficiency of Character Replacement

Because strings are immutable, replacing characters involves creating new strings. This can become inefficient for large strings that require multiple modifications, as each replacement results in a new string.

Example:

```javascript
let str = "JavaScript";
str = str.replace("Java", "Type"); // Creates a new string "TypeScript"
console.log(str); // Output: "TypeScript"
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
TypeScript
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Escape characters

The escape characters are used to include special characters in strings. Escape sequences begin with a backslash (\) followed by a character or combination of characters to represent certain special characters or formatting.

There are many escape characters, most commonly used escape characters are listed below:-

S. No	Escape Sequence	Description	Example	Output
01.	\'	Single quote	'It\'s a great day!'	It's a great day!
02.	\"	Double quote	"He said, \"Hello!\""	He said, "Hello!"
03.	\\	Backslash	"This is a backslash: \\"	This is a backslash: \
04.	\n	New line	"Hello\nWorld"	Hello World
05.	\r	Carriage return	"Hello\rWorld"	World (replaces "Hello" with "World")
06.	\t	Tab	"Name:\tJohn Doe"	Name: John Doe
07.	\b	Backspace	"Hello\bWorld"	HellWorld
08.	\f	Form feed	"Hello\fWorld"	Hello World (form feed effect depends on environment)
09.	\v	Vertical tab	"Hello\vWorld"	Hello World (vertical tab effect depends on environment)
10.	\u00A9	Unicode character (hex)	"\u00A9 2024"	© 2024
11.	\xA9	ASCII character (hex)	"\xA9"	©

String Methods

String methods are built-in functions that you can use to manipulate and work with strings. These methods perform various tasks, like modifying, searching, or extracting information from strings.

Types of String Methods

There are various types of string methods available to perform read, update , delete & manipulate operations as given below:-

Length

The length method in the string method returns the total number of characters in a string, including spaces and punctuation.

Example:

```javascript
let text = "JavaScript";
console.log(text.length); // Output: 10
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
10
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

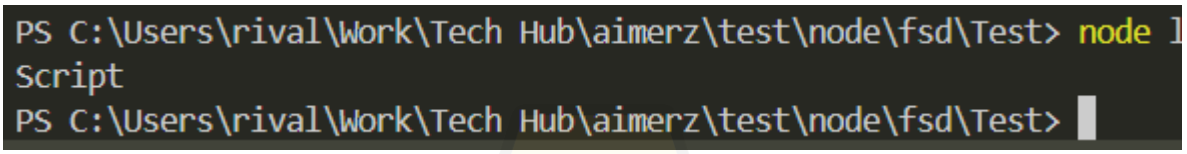
Slice

The slice method in the string methods extracts a section of the string, defined by start and end indexes, and returns a new string.

Example:

```javascript
let text = "JavaScript";
console.log(text.slice(4, 10)); // Output: "Script"
```

OUTPUT:



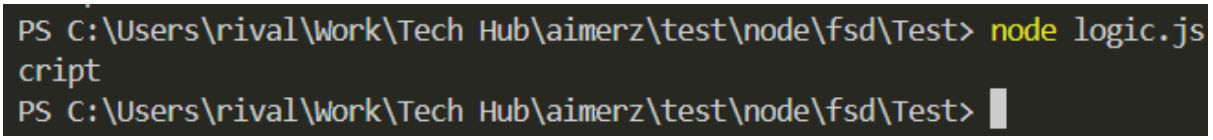
Substring

The Substring method in the string method is similar to slice(), but does not accept negative indexes; used to extract a substring based on start and end indexes.

Example:

```javascript
let text = "JavaScript";
console.log(text.substring(5, 10)); // Output: "Script"
```

OUTPUT:



substr

The substr method in the string methods extracts a part of the string starting from an index and for a defined length.

Example:

```javascript
let text = "JavaScript";
console.log(text.substring(5, 10)); // Output: "cript"
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
cript
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Replace

The Replace method in the string methods replaces only the first occurrence of a substring with a replacement string.

Example:

```javascript
let greeting = "Hello World";
console.log(greeting.replace("World", "Java")); // Output: "Hello Java"
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
Hello Java
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

replaceAll

The replaceAll method in the string methods replaces all occurrences of a specified substring with another, useful for character replacements

Example:

```javascript
let greeting = "Hello World";
console.log(greeting.replaceAll("l", "L")); // Output: "HeLLo WorLd"
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
HeLLo WorLd
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

toUpperCase

It is used to change the case of all alphabetic characters to upper case.

Example:

```javascript
let greeting = "Hello World";
console.log(greeting.toUpperCase());
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
HELLO WORLD
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

toLowerCase

It is used to change the case of all alphabetic characters to lower case.

Example:

```javascript
let greeting = "Hello World";
console.log(greeting.toLowerCase());
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
hello world
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Concat

It combines multiple strings into one, often cleaner than using + for string concatenation.

Example:

```javascript
let hello = "Hello";
console.log(hello.concat(" ", "World")); // Output: Hello World
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
Hello World
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

split

The split() splits a string into an array of substrings based on a specified separator.

Example:

```javascript
let csv = "a,b,c";
console.log(csv.split(",")); // Output: ["a", "b", "c"]
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
[ 'a', 'b', 'c' ]
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

indexOf

The indexOf() and lastIndexOf() finds the first or last index of a substring in a string, returning -1 if not found.

Example:

```javascript
let text = "Hello";
```

```
console.log(text.indexOf("e"));
console.log(text.lastIndexOf("l"));
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
1
3
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

startsWith & endsWith

startsWith() and endsWith() checks whether a string starts or ends with a particular substring.

Example:

```
JavaScript
let text = "Hello";
console.log(text.startsWith("e"));
console.log(text.endsWith("o"));
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node lo
false
true
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Search

Searches for a regular expression within a string and returns the index of the match.

Example:

```
JavaScript
let text = "Hello";
```

```
console.log(text.search(/llo/));
```

Output:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
2
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

Trim

The trim() removes whitespace from the beginning and end of a string, without affecting spaces inside.

Example:

```javascript
let str = "  Hello World!  ";
let trimmedStr = str.trim();  // "Hello World!"
console.log(str);
console.log(trimmedStr);
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
  Hello World!
Hello World!
```

charAt & at

charAt() and at() returns the character at a specific index, with at() allowing negative indexes.

Example:

```javascript
let spaced = "Hello World";
console.log(spaced.charAt(6));
console.log(spaced.at(-1));
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
W
d
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

charCodeAt

The charCodeAt() returns the Unicode value of a character at a specified index.

Example:

```javascript
let spaced = "Hello World";
console.log("W".charCodeAt(0)); // Output: 65
```

OUTPUT:

```
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test> node logic.js
87
PS C:\Users\rival\Work\Tech Hub\aimerz\test\node\fsd\Test>
```

THANK YOU

AIMERZ.ai
Aim.Act.Achieve

Stay updated with us!

Vishwa Mohan

Vishwa Mohan

vishwa.mohan.singh

Vishwa Mohan