STRING

1. Basic Natute

Strings are sequences of characters, typically used to represent text in programming. Here are some basic characteristics of strings:

1. **Immutable** : In many programming languages, strings are immutable, meaning that once a string is created, its contents cannot be changed. However, operations on strings often return new strings rather than modifying the original string.
2. **Sequence** : Strings are considered sequences of characters, where each character occupies a specific position within the string. This allows for various operations such as accessing individual characters, slicing, concatenating, and iterating over the characters in the string.
3. **Encoding** : Strings can be encoded using different character encoding schemes, such as ASCII, UTF-8, UTF-16, etc. These encoding schemes define how characters are represented as binary data.
4. **Length** : The length of a string refers to the number of characters it contains. This length can be obtained using built-in functions or properties provided by

programming languages.

5. **Concatenation** : Strings can be concatenated, or joined together, to create longer strings. This is often done using the + operator or specific string concatenation functions provided by programming languages.
6. **Indexing and Slicing** : Individual characters within a string can be accessed using zero-based indexing. Additionally, substrings can be obtained by slicing the string using start and end indices.
7. **Comparison** : Strings can be compared for equality or inequality, as well as lexicographically (based on alphabetical order). This allows for sorting and searching operations on strings.

Overall, strings are fundamental data types in programming and are widely used for representing and manipulating text-based data.

2. String indexing

String indexing refers to the process of accessing individual characters within a string using their position or index. In most programming languages, strings are zero-indexed, meaning that the first character of the string is located at index 0, the second character at index 1, and so on.

Here's an example to illustrate string indexing:

```
const str = "Hello";

// Accessing individual characters
console.log(str[0]); // Output: "H"
console.log(str[1]); // Output: "e"
console.log(str[2]); // Output: "l"
console.log(str[3]); // Output: "l"
console.log(str[4]); // Output: "o"

// Accessing characters using negative indices (starting from the end)
console.log(str[-1]); // Output: undefined (Negative indices are not valid)
console.log(str[-2]); // Output: undefined
console.log(str[-3]); // Output: undefined
console.log(str[-4]); // Output: undefined
console.log(str[-5]); // Output: undefined

// Accessing characters using indices beyond the length of the string
console.log(str[5]); // Output: undefined (Index out of range)
console.log(str[10]); // Output: undefined
```

=> In the example above, the string "Hello" is indexed from 0 to 4, with each index representing a specific character in the string.

=> It's important to note that attempting to access an index outside the range of the string will result in an error or undefined behavior, depending on the programming language. Additionally, some languages may support negative indexing, where -1 represents the last character of the string, -2 represents the second-to-last character, and so on.

3. String Iteration

String iteration refers to the process of traversing or looping through each character in a string to perform some operation on each character individually or on substrings of the string. Iterating over a string is commonly done using loops or higher-order functions provided by programming languages.

Here are some common methods for iterating over a string:

1.For Loop : Using a traditional `for` loop to iterate through each character in the string.

```
const str = "Hello";

for (let i = 0; i < str.length; i++) {
  console.log(str[i]); // Output: "H", "e", "l", "l", "o"
}
```

2.**For...of Loop** : Using a `for...of` loop to iterate through each character in the string directly.

```
const str = "Hello";

for (const char of str) {
  console.log(char); // Output: "H", "e", "l", "l", "o"
}
```

3.**Array Iteration Methods** : Converting the string to an array of characters and then using array iteration methods like `forEach`, `map`, `filter`, etc.

```
const str = "Hello";

Array.from(str).forEach(char => {
  console.log(char); // Output: "H", "e", "l", "l", "o"
});
```

=> String iteration is useful for tasks such as searching for specific characters, transforming characters, or performing any operation that involves processing individual characters or substrings within a string.

4. String Concatenation

String concatenation refers to the process of combining two or more strings into a single string. In most programming languages, string concatenation can be achieved using various methods or operators.

Here are some common ways to concatenate strings:

1.**Using the `+` Operator** : In many programming languages, the `+` operator is used to concatenate strings.

```
const str1 = "Hello";
const str2 = "World";
const result = str1 + " " + str2;
console.log(result); // Output: "Hello World"
```

2.**Using the `concat()` Method** : Some languages provide a `concat()` method to concatenate strings.

```
const str1 = "Hello";
const str2 = "World";
const result = str1.concat(" ", str2);
console.log(result); // Output: "Hello World"
```

3.**Template Literals** : Template literals, also known as template strings, allow for easy string interpolation and concatenation using backticks (`).

```
const str1 = "Hello";
const str2 = "World";
const result = `${str1} ${str2}`;
console.log(result); // Output: "Hello World"
```

=> String concatenation is a common operation when building strings dynamically, such as constructing messages, generating HTML or XML, or formatting output. It's important to note that repeated concatenation of large numbers of strings can be inefficient due to the creation of multiple intermediate string objects. In such cases, using methods like `join()` or building an array of strings before joining them together may be more efficient.

5. Subsstring extraction

Substring extraction refers to the process of extracting a portion of a string based on specified start and end indices or based on a substring pattern. This allows you to obtain a smaller string from a larger string.

Here are common ways to extract substrings from a string:

1.Using `substring()` Method : The `substring()` method extracts the characters from a string between two specified indices (start and end), or to the end of the string if no end index is provided.

```
const str = "Hello World";
const substring1 = str.substring(0, 5); // Extracts characters from index 0
to 4 (excluding 5)
console.log(substring1); // Output: "Hello"

const substring2 = str.substring(6); // Extracts characters from index 6 to
the end of the string
console.log(substring2); // Output: "World"
```

2.**Using `substr()` Method** : The `substr()` method extracts a specified number of characters from a string, starting at a specified index.

```
const str = "Hello World";
const substring1 = str.substr(0, 5); // Extracts 5 characters starting from
index 0
console.log(substring1); // Output: "Hello"
```

```
const substring2 = str.substr(6); // Extracts characters from index 6 to the
end of the string
console.log(substring2); // Output: "World"
```

3.**Using `slice()` Method** : The `slice()` method extracts a section of a string and returns it as a new string, specifying the start and end indices.

```
const str = "Hello World";
const substring1 = str.slice(0, 5); // Extracts characters from index 0 to 4
(excluding 5)
console.log(substring1); // Output: "Hello"

const substring2 = str.slice(6); // Extracts characters from index 6 to the
end of the string
console.log(substring2); // Output: "World"
```

=> Substring extraction is useful for tasks such as parsing data from strings, extracting specific portions of text, or manipulating string segments independently. It provides flexibility in working with strings by allowing you to focus on relevant parts of the data.

6. String Case Conversion

String case conversion refers to the process of changing the case (uppercase or lowercase) of the characters within a string. This operation is often needed in various text processing tasks to standardize or manipulate the case of strings.

Here are common methods to convert the case of strings:

1. **To Uppercase (`toUpperCase()`)** : This method converts all characters in a string to uppercase.

   ```
   const str = "hello world";
   const upperCaseStr = str.toUpperCase();
   console.log(upperCaseStr); // Output: "HELLO WORLD"
   ```

2. **To Lowercase (`toLowerCase()`)** : This method converts all characters in a string to lowercase.

   ```
   const str = "HELLO WORLD";
   const lowerCaseStr = str.toLowerCase();
   ```

```
console.log(lowerCaseStr); // Output: "hello world"
```

3. **To Title Case (Custom Function)** : Title case refers to capitalizing the first letter of each word in a string. While there isn't a built-in method for this in JavaScript, you can create a custom function to achieve it.

```
function toTitleCase(str) {
    return str.toLowerCase().replace(/(?:^|\s)\w/g, function(match) {
        return match.toUpperCase();
    });
}

const sentence = "hello world";
const titleCaseSentence = toTitleCase(sentence);
console.log(titleCaseSentence); // Output: "Hello World"
```

=> String case conversion is commonly used for formatting text, normalizing input data, or enforcing specific conventions in string representations. Depending on the requirements of your application, you can choose the appropriate method to convert the case of strings accordingly.

7. String Triming

String trimming is the process of removing whitespace characters from the beginning and/or end of a string. Whitespace characters include spaces, tabs, and newline characters. Trimming is often used to clean up user input or normalize data before processing.

In JavaScript, you can use the `trim()` method to trim whitespace from both ends of a string. Here's how it works:

```
const str = "   Hello, world!   ";
const trimmedStr = str.trim();

console.log(trimmedStr); // Output: "Hello, world!"
```

In the example above:

- The original string `str` contains leading and trailing whitespace.

- The `trim()` method is called on the string, which returns a new string with leading and trailing whitespace removed.
- The resulting string `trimmedStr` contains the original content of the string without any leading or trailing whitespace.

The `trim()` method is useful for removing unnecessary whitespace, such as when processing user input from form fields or cleaning up data from external sources.

It's important to note that the `trim()` method only removes leading and trailing whitespace. If you need to remove whitespace from other parts of the string, you can use regular expressions or other string manipulation techniques. For example, you can use `replace()` with a regular expression to remove all whitespace characters:

```
const str = "   Hello, world!   ";
const trimmedStr = str.replace(/^\s+|\s+$/g, "");

console.log(trimmedStr); // Output: "Hello,world!"
```

In this example, the regular expression `^\s+|\s+$` matches one or more whitespace characters (`\s+`) at the beginning (`^`) or end (`$`) of the string, and replaces them with an empty string.

8. String Split And Join (With Array)

String splitting and joining are common string manipulation operations that involve splitting a string into an array of substrings or joining an array of substrings into a single string.

1.**String Splitting (With Array)** :

- The `split()` method in JavaScript is used to split a string into an array of substrings based on a specified separator.
- The method takes an optional parameter that specifies the separator to use for splitting the string. If no separator is provided, the string is split into individual characters.
- Here's an example of using `split()` to split a string into an array of words based on whitespace:

```
const sentence = "The quick brown fox";
const words = sentence.split(" ");

console.log(words); // Output: ["The", "quick", "brown", "fox"]
```

2.**Array Joining (With String)** :

- The `join()` method in JavaScript is used to join the elements of an array into a single string using a specified separator.
- The method takes an optional parameter that specifies the separator to use for joining the elements. If no separator is provided, a comma is used by default.
- Here's an example of using `join()` to join an array of words into a single string with spaces between them:

```
const words = ["The", "quick", "brown", "fox"];
const sentence = words.join(" ");

console.log(sentence); // Output: "The quick brown fox"
```

Combining string splitting and joining allows you to manipulate strings and arrays in various ways. For example, you can split a string into an array, modify the elements of the array, and then join the array back into a single string.

Here's an example of using string splitting and joining together to reverse the words in a sentence:

```
const sentence = "The quick brown fox";
const reversed = sentence.split(" ").reverse().join(" ");

console.log(reversed); // Output: "fox brown quick The"
```

=> In this example, the sentence is split into an array of words using `split()`, the order of the words is reversed using `reverse()`, and then the reversed words are joined back into a single string using `join()`.

9. String Replace Character

In JavaScript, you can replace characters within a string using various methods, such as `replace()` method or regular expressions. Below are examples of how to replace characters within a string:

1.Using `replace()` method:

The `replace()` method is used to search for a specified pattern (or substring) in a string and replace it with another string or a function that returns the replacement string. It only replaces the first occurrence of the pattern by default. To replace all occurrences, you can use a regular expression with the global (`g`) flag.

```
const str = "Hello World!";
const replacedStr = str.replace("World", "Universe");

console.log(replacedStr); // Output: "Hello Universe!"
```

2.Using regular expressions:

Regular expressions offer more flexibility when replacing characters in a string. You can use the `replace()` method with a regular expression pattern to replace characters that match the pattern.

```
const str = "Hello World!";
const replacedStr = str.replace(/o/g, "0");

console.log(replacedStr); // Output: "Hell0 W0rld!"
```

3.Using a combination of `split()` and `join()`:

Another approach is to split the string into an array of characters, manipulate the array (e.g., replace characters), and then join the array back into a string.

```
const str = "Hello World!";
const charArray = str.split("");
charArray[4] = "y"; // Replace 'o' with 'y'
const replacedStr = charArray.join("");

console.log(replacedStr); // Output: "HellY World!"
```

=> All these methods allow you to replace characters within a string in JavaScript. Choose the one that best fits your specific use case and requirements.

10. String Reversing

In JavaScript, you can reverse a string using various methods. Here are a few commonly used approaches:

1.**Using Built-in Methods** :

JavaScript provides the `split()`, `reverse()`, and `join()` methods to reverse a string:

```
const str = "Hello World!";
const reversedStr = str.split("").reverse().join("");

console.log(reversedStr); // Output: "!dlroW olleH"
```

2. **Using a Loop** :

   You can manually reverse a string using a loop to iterate through each character:

   ```
   function reverseString(str) {
     let reversedStr = "";
     for (let i = str.length - 1; i >= 0; i--) {
       reversedStr += str[i];
     }
     return reversedStr;
   }

   const str = "Hello World!";
   const reversedStr = reverseString(str);

   console.log(reversedStr); // Output: "!dlroW olleH"
   ```

3. **Using Recursion** :

   Recursion can also be used to reverse a string:

   ```
   function reverseString(str) {
     if (str === "") {
       return "";
     } else {
       return reverseString(str.substr(1)) + str.charAt(0);
     }
   }
   ```

```
const str = "Hello World!";
const reversedStr = reverseString(str);

console.log(reversedStr); // Output: "!dlroW olleH"
```

=> This approach works by recursively reversing the substring starting from the second character and appending the first character to the end.

=> Each of these methods provides a way to reverse a string in JavaScript. Choose the one that best fits your specific use case and coding style.

11.1 String Plindrome

A palindrome is a word, phrase, number, or other sequence of characters that reads the same forward and backward. For example, "radar", "level", and "12321" are all palindromes.

To check if a string is a palindrome in JavaScript, you can compare the original string with its reverse. If they are the same, the string is a palindrome.

Here's how you can implement this check:

```
function isPalindrome(str) {
  // Remove non-alphanumeric characters and convert to lowercase
  const alphanumericStr = str.toLowerCase().replace(/[^a-z0-9]/g, "");

  // Reverse the string
  const reversedStr = alphanumericStr.split("").reverse().join("");

  // Compare original and reversed strings
  return alphanumericStr === reversedStr;
}

// Test cases
console.log(isPalindrome("radar")); // true
console.log(isPalindrome("level")); // true
console.log(isPalindrome("12321")); // true
console.log(isPalindrome("A man, a plan, a canal, Panama!")); // true
console.log(isPalindrome("hello")); // false
```

In this implementation:

1. We first remove non-alphanumeric characters and convert the string to lowercase using a regular expression.

2. We then reverse the string by splitting it into an array of characters, reversing the array, and joining the characters back into a string.
3. Finally, we compare the original and reversed strings. If they are equal, the original string is a palindrome.

This function correctly identifies palindromes regardless of case and non-alphanumeric characters. You can use it to check whether a given string is a palindrome in JavaScript.

11.2 Number Palindrome

To check if a number is a palindrome in JavaScript, you can convert the number to a string and then use the same logic as checking a palindrome string. Here's how you can implement it:

```
function isPalindrome(num) {
  // Convert number to string
  const str = num.toString();

  // Reverse the string
  const reversedStr = str.split("").reverse().join("");

  // Compare original and reversed strings
  return str === reversedStr;
}

// Test cases
console.log(isPalindrome(12321)); // true
console.log(isPalindrome(12345)); // false
```

In this implementation:

1. We convert the number to a string using `toString()` method.
2. We then reverse the string by splitting it into an array of characters, reversing the array, and joining the characters back into a string.
3. Finally, we compare the original and reversed strings. If they are equal, the original number is a palindrome.

This function correctly identifies palindromes regardless of the number's digits. You can use it to check whether a given number is a palindrome in JavaScript.

12. Anagrams

An anagram is a word or phrase formed by rearranging the letters of another word or phrase, typically using all the original letters exactly once. For example, "listen" and "silent" are anagrams of each other.

To check if two strings are anagrams in JavaScript, you can compare their sorted forms. If the sorted forms of both strings are equal, then the strings are anagrams.

Here's how you can implement this:

```
function isAnagram(s, t) {
    // If the lengths of the two strings are different, they cannot be
anagrams
    if (s.length !== t.length) {
        return false;
    }

    // Create character frequency maps for both strings
    const frequencyS = {};
    const frequencyT = {};

    // Update frequency map for string s
    for (let char of s) {
        frequencyS[char] = (frequencyS[char] || 0) + 1;
    }

    // Update frequency map for string t
    for (let char of t) {
        frequencyT[char] = (frequencyT[char] || 0) + 1;
    }

    // Compare character frequencies in both maps
    for (let key in frequencyS) {
        if (frequencyS[key] !== frequencyT[key]) {
            return false;
        }
    }

    return true;
}

// Test cases
console.log(isAnagram("anagram", "nagaram")); // Output: true
console.log(isAnagram("rat", "car")); // Output: false
```

In this implementation:

1. We first remove non-alphanumeric characters and convert both strings to lowercase using regular expressions.

2. We then split the strings into arrays of characters, sort the arrays, and join the characters back into strings.
3. Finally, we compare the sorted forms of the strings. If they are equal, the original strings are anagrams.

This function correctly identifies anagrams regardless of the case and non-alphanumeric characters. You can use it to check whether two given strings are anagrams in JavaScript.

13. Longest SubString

To find the longest substring without repeating characters in a given string, you can use the sliding window technique. The sliding window technique involves maintaining a window of characters and moving it across the string to find the longest substring that meets certain conditions.

Here's how you can implement it in JavaScript:

```
function lengthOfLongestSubstring(s) {
  let maxLength = 0;
  let start = 0;
  const charIndexMap = {};

  for (let end = 0; end < s.length; end++) {
    const currentChar = s[end];
    if (charIndexMap[currentChar] >= start) {
      start = charIndexMap[currentChar] + 1;
    }
    charIndexMap[currentChar] = end;
    maxLength = Math.max(maxLength, end - start + 1);
  }

  return maxLength;
}

// Test case
const s = "abcabcbb";
console.log(lengthOfLongestSubstring(s)); // Output: 3
```

In this implementation:

- We use a map (`charIndexMap`) to store the index of each character encountered.
- We initialize `start` to 0 and iterate through the string using `end` as the end index of the current substring.

- If the current character is already in the map and its index is greater than or equal to `start`, we update `start` to the index of the repeated character plus one.
- We update the index of the current character in the map.
- We update `maxLength` to the maximum length of the current substring.
- Finally, we return `maxLength`, which represents the length of the longest substring without repeating characters.

This algorithm has a time complexity of O(n), where n is the length of the input string, as it only iterates through the string once.