SUNSTONE

JavaScript

LECTURE 13 NOTES

Regular Expressions

- **What are Regular Expressions?**

  - In JavaScript, regular expressions (often abbreviated as regex or regexp) are powerful tools for pattern matching and manipulating strings. They are represented by a sequence of characters that define a search pattern. Regular expressions are commonly used for tasks such as validating input, extracting specific information from strings, and performing complex string manipulations.

  - In JavaScript, regular expressions are created using the RegExp constructor or by using a literal syntax enclosed in forward slashes (/).

    SYNTAX:

    ```
    var regex = /pattern/;
    ```

  - In the above example, pattern represents the actual pattern you want to match or search for.

  - Regular expressions consist of various special characters and metacharacters that define different patterns.

  - Here are some of the basic patterns and metacharacters commonly used in JavaScript regular expressions:

➢ **Literal Characters:** Regular characters in the pattern match themselves.

For example, the pattern /hello/ would match the string “hello”.

➢ **Metacharacters:** These characters have special meaning in regular expressions. Some commonly used metacharacters include:

- **. (dot):** Matches any single character except a newline.
- **^ (caret):** Matches the start of a string.
- **$ (dollar sign):** Matches the end of a string.
- *** (asterisk):** Matches zero or more occurrences of the preceding character or group.
- **+ (plus):** Matches one or more occurrences of the preceding character or group.

- **? (question mark):** Matches zero or one occurrence of the preceding character or group.
- **[ ] (square brackets):** Matches any single character within the brackets.
- **| (pipe):** Acts as an OR operator between two or more patterns.

➢ **Character Classes:** Character classes allow you to specify a set of characters to match. Some common character classes include:

- **\d:** Matches any digit (0-9).
- **\w:** Matches any word character (a-z, A-Z, 0-9, or underscore _).
- **\s:** Matches any whitespace character (space, tab, newline, etc.).
- **\D, \W, \S:** Negated versions of \d, \w, \s.

➢ **Quantifiers:** Quantifiers define the number of occurrences to match. Some commonly used quantifiers include:

- **{n}:** Matches exactly n occurrences of the preceding character or group.
- **{n,}:** Matches at least n occurrences of the preceding character or group.
- **{n,m}:** Matches between n and m occurrences of the preceding character or group.

- These are just a few examples of the syntax and patterns used in JavaScript regular expressions. Regular expressions can be much more complex, allowing for advanced pattern matching and manipulation of strings.
- To use regular expressions in JavaScript, you can use methods like `test()`, `match()`, `replace()`, and `split()` that are available on strings or the RegExp object to perform various operations based on the defined patterns.

  For example,

  Creating a regular expression

  You can construct a regular expression in one of two ways:

1. **Using a regular expression literal,** which consists of a pattern enclosed between slashes, as follows:

```
const re = /ab+c/;
```

Regular expression literals provide compilation of the regular expression when the script is loaded. If the regular expression remains constant, using this can improve performance.

2. **Calling the constructor function of the RegExp object,** as follows:

```
const re = new RegExp("ab+c");
```

Using the constructor function provides runtime compilation of the regular expression. Use the constructor function when you know the regular expression pattern will be changing, or you don't know the pattern and are getting it from another source, such as user input.

- **Metacharacters and Quantifiers**

➢ **Metacharacters**

- These characters have special meaning in regular expressions.

1. **. (dot):** Matches any single character except a newline.

For example:

```
var regex = /h.t/;  // Matches "hat", "hot", "hit", etc.
```

2. **^ (caret):** Matches the start of a string.

For example:

```
var regex = /^hello/;   // Matches strings starting with "hello"
```

3. **$ (dollar sign):** Matches the end of a string.

For example:

```
var regex = /world$/;  // Matches strings ending with "world"
```

4. *** (asterisk):** Matches zero or more occurrences of the preceding character or group.

   For example:
   ```
   var regex = /ab*c/;  // Matches "ac", "abc", "abbc", etc.
   ```

5. **+ (plus):** Matches one or more occurrences of the preceding character or group.

   For example:
   ```
   var regex = /ab+c/;  // Matches "abc", "abbc", "abbbc", etc.
   ```

6. **? (question mark):** Matches zero or one occurrence of the preceding character or group.

   For example:
   ```
   var regex = /colou?r/;  // Matches "color" and "colour"
   ```

7. **[ ] (square brackets):** Matches any single character within the brackets.

   For example:
   ```
   var regex = /[aeiou]/;  // Matches any vowel character
   ```

8. **| (pipe):** Acts as an OR operator between two or more patterns.

   For example:
   ```
   var regex = /apple|orange/;  // Matches "apple" or "orange"
   ```

➢ **Quantifiers:**

- Quantifiers define the number of occurrences to match.

1. **{n}:** Matches exactly n occurrences of the preceding character or group.

   For example:
   ```
   var regex = /a{3}/;  // Matches "aaa"
   ```

2. **{n,}:** Matches at least n occurrences of the preceding character or group.

   For example:
   ```
   var regex = /a{2,}/;  // Matches "aa", "aaa", "aaaa", etc.
   ```

3. **{n,m}:** Matches between n and m occurrences of the preceding character or group.

   For example:

   ```
   var regex = /a{2,4}/;  // Matches "aa", "aaa", "aaaa"
   ```

- **Here's an example that combines multiple metacharacters and quantifiers:**

  ```
  var regex = /^[\w.-]+@\w+\.[a-z]{2,}$/;
  ```

- This regular expression is commonly used to validate email addresses.
- In the example above, the pattern matches an email address that starts with one or more word characters, followed by the @ symbol, then one or more word characters, a dot, and finally, a two or more lowercase letter domain extension.

• Capturing and Non-capturing Groups

- In JavaScript regular expressions, capturing and non-capturing groups are used to group and capture parts of a matched pattern. They are denoted by parentheses () in the regular expression pattern.

➢ Capturing Groups

- Capturing groups are used to capture and store the matched portion of the string for future reference.
- They are assigned a numerical index starting from 1, based on their position in the regular expression.
- Captured groups can be accessed using methods like `exec()` or `match()`.
- They are also used in replacement patterns with `replace()`.

- Here's an example of a capturing group:

```
var regex = /(\d{2})-(\d{2})-(\d{4})/;   // Matches dates in the format DD-MM-YYYY

var str = 'Today is 22-06-2023';

var match = regex.exec(str);

console.log(match[0]);                   // 22-06-2023

console.log(match[1]);                   // 22

console.log(match[2]);                   // 06

console.log(match[3]);                   // 2023
```

➢ Non-capturing Groups

- Non-capturing groups are used to group parts of the pattern without capturing them.
- They are denoted by (?:...) syntax, where ?: indicates a non-capturing group.
- Non-capturing groups are useful when you want to group a part of the pattern but don't need to store the matched value separately.
- Non-capturing groups don't affect the indexing of captured groups.
- Here's an example of a non-capturing group:

```
var regex = /(?:https?:\/\/)?(?:www\.)?(\w+\.\w+)/;   // Matches URLs without capturing the protocol or "www" subdomain

var str = 'Visit my website at https://www.example.com';

var match = regex.exec(str);

console.log(match[0]);                   // www.example.com

console.log(match[1]);                   // example.com
```

- In the above example, the non-capturing group (?:https?:\/\/)? matches an optional protocol (http:// or https://) without capturing it. The non-capturing group (?:www\.)? matches an optional "www" subdomain without capturing it. The capturing group (\w+\.\w+) captures the main domain name.
- Capturing and non-capturing groups provide flexibility in organizing and manipulating matches within regular expressions, depending on your specific requirements.

• Regular Expression Methods and Flags

- JavaScript provides several methods and flags that can be used with regular expressions to perform various operations.
- Let's explore the most commonly used methods and flags:

➢ Regular Expression Methods

1. **test():** Tests if a pattern matches a string and returns true or false.

   Example:

```
var regex = /hello/;
console.log(regex.test("hello world"));  // true
```

2. **exec():** Searches a string for a match against a pattern and returns an array containing the matched results.

   Example:

```
var regex = /\d+/;
var match = regex.exec("I have 42 apples");
console.log(match[0]);  // 42
```

3. **match():** Retrieves the matches when a string matches a pattern and returns an array or null.

   Example:

```
var regex = /\d+/;
var matches = "I have 42 apples".match(regex);
```

```
console.log(matches[0]);  // 42
```

4. **search():** Searches a string for a specified pattern and returns the index of the first match or -1 if not found.

   Example:

   ```
   var regex = /world/;
   console.log("Hello world".search(regex));  // 6
   ```

5. **replace():** Searches a string for a specified pattern and replaces the matches with a replacement string.

   Example:

   ```
   var regex = /John/;
   var str = "Hello John!";
   var newStr = str.replace(regex, "Alice");
   console.log(newStr);  // Hello Alice!
   ```

6. **split():** Splits a string into an array of substrings based on a specified separator pattern.

   Example:

   ```
   var regex = /\s+/;
   var str = "Hello world";
   var array = str.split(regex);
   console.log(array);  // ["Hello", "world"]
   ```

➢ Regular Expression Flags

1. **i (ignore case):** Performs a case-insensitive match.

   Example:

   ```
   var regex = /hello/i;
   console.log(regex.test("Hello World"));  // true
   ```

2. **g (global):** Performs a global match, finding all occurrences instead of stopping after the first match.

   Example:

   ```
   var regex = /apple/g;
   var str = "I have an apple and another apple";
   var matches = str.match(regex);
   console.log(matches);  // ["apple", "apple"]
   ```

3. **m (multiline):** Allows matching over multiple lines using the ^ and $ anchors.

   Example:

   ```
   var regex = /^start.*end$/m;
   var str = "start\n this is the end";
   console.log(regex.test(str));  // true
   ```

- These are some of the commonly used methods and flags for regular expressions in JavaScript. Regular expressions provide powerful pattern-matching capabilities, and these methods and flags allow you to utilize them effectively in your code.