Lesson:

Regex in Javascript

PW SKILLS

Topics to be covered

1. Introduction to Regex
2. Creating a regex

Introduction to Regex

Regex stands for the regular expression. It is a very powerful tool for pattern matching and text manipulation in javascript and in many other languages also.
Regex allows us to search, extract, replace and validate strings based on specific patterns.

Creating a regex

In javascript, we can create a regular expression by using the literal syntax. The literal syntax is enclosed within forward slashes(/..../) and the other matches will be in-between slashes to which we want to write as our regex.

Syntax

```
JavaScript
const regex = /pattern/;
```

In the above example, we had created a variable named regex and our pattern will be enclosed in between the slashes.

Matching text using regex
The most basic operation we can do using regex is to find the match within a string.
For this we can use any of these two methods as per our requirements-

1. test()
We can use the test method to check that the string has the regex string in it or not. It will return true if the string will have the regex match in it else will return false.

```
JavaScript
const myString = "Welcome to PW Skills";
const regex1 = /pw/;
const regex2 = /PW/;
console.log(typeof regex1);
const output1 = regex1.test(myString);
const output2 = regex2.test(myString);
console.log(output1, output2);

// output
// object
// false true
```

In the above code, we have a myString which contains a string and has two regexes named regex1 and regex2 enclosed by slashes and contains the regex pattern in it. So both of the regex is checking for the pattern that it is there in the myString or not and as per output we can see that the regex1 is false because 'pw' is not in the string as it is checking for the exact match with case sensitivity also while in the regex2 we are getting true because the pattern 'PW' is inside the myString.
Also apart from these two we had also logged the type of regex and we are getting an object, so regex is also of type object in javascript.

2. match()
Similar to the test method it is also used to check that the string contains the regex pattern in it or not but instead of returning a boolean value it returns us the exact value to which it is found in the string.

```javascript
const myString = "Welcome to PW Skills";
const regex1 = /pw/;
const regex2 = /PW/;
const output1 = myString.match(regex1);
const output2 = myString.match(regex2);
console.log(output1, output2);

// output
// object
// null [ 'PW', index: 11, input: 'Welcome to PW Skills', groups: undefined ]
```

As in the above code we can see that we had used the match method to check that the regex pattern is there in the string or not.
Hence in the output1 we are getting as it is not in the string while the output2 is returning us an array containing the searched string with its index, input and groups.

1. 'PW'

It is the matched substring which matches the regex pattern.

1. index:11

It is the starting index of the matched substring in its original string.

1. input

It is the original string passed to check against the regex.

1. groups

This property is related to the capturing groups in regex to which we will discuss in detail while moving further in this class.

Note:
The test method is used on regex while the match method is available as a string method.

Character Classes

Character classes allow us to match specific sets of characters against any string which is quite useful in conditions like matching password complexity, mobile number checking etc.

Some common characters classes are-

1. [abc]

This will match for any character either a, b or c in a string.

1. [0-9]

This will match for any digits in a string.

1. [A-Z]

This will match for any character in uppercase from A to Z, similarly we can do for lowercase letters also.

Example

```
JavaScript
const myString = "PW Skills";
const pinCode = "256314";

console.log(myString.match(/[abc]/));
console.log(myString.match(/[a-z]/));
console.log(myString.match(/[A-Z]/));
console.log(pinCode.match(/[9801]/));
console.log(pinCode.match(/[0-9]/));

// output
// null
// [ 'k', index: 4, input: 'PW Skills', groups: undefined ]
// [ 'P', index: 0, input: 'PW Skills', groups: undefined ]
// [ '1', index: 4, input: '256314', groups: undefined ]
// [ '2', index: 0, input: '256314', groups: undefined ]
```

Example Explanation

First Console: It is null as in the string there is no character a, b or c.

Second Console: We are checking for any lowercase character in between a to z in the string, so getting index 4 and value as ‘k’ as it is the first lowercase character in the string.

Third Console: Similar to the second console it is giving us index 0 and character ‘P’ as it is the first upper case character in the string in between A to Z.

Fourth Console: We are matching for character 9, 8, 0 or 1 in the pin code and it found 1 at the index 4 which is the first match with our given pattern in regex.

Fifth Console: We are getting index 0 as value 2 as it is the first character matched in between 0 to 9.

Modifiers

Modifiers are used after the closing slash and it is used to modify the behavior of a regex.
Some common modifiers are-

1. g

It is used to perform the global match instead of stopping at the first match as we had seen in the above examples.

```
JavaScript
const myString = "PW Skills";
console.log(myString.match(/[a-z]/g));

// output
// ["k", "i", "l", "l", "s"]
```

In the above example, we are getting the characters which are in between lowercase a to z in an array.

2. i

As we had seen that the regex patterns are case sensitive in above examples of matching text, so if we want to match any text without being worried about its case, then we will use this modifier.

```
JavaScript
const myString = "PW Skills";
console.log(myString.match(/p/));
console.log(myString.match(/p/i));

// output
// null
// [ 'P', index: 0, input: 'PW Skills', groups: undefined ]
```

As we can see in the above example, if we are not using the modifier then it is giving us null as there is no matching character in it, but with the modifier it is giving us the array as the character is available at the first index but in uppercase.

Combining Modifiers

We can combine the modifier to get the desired result as in the below example -

```
JavaScript
const myString = "PW Skills Pvt Ltd";
console.log(myString.match(/[pL]/gi));

// output
// [ 'P', 'l', 'l', 'P', 'L' ]
```

Now we are getting all the matches without checking for the case sensitivity.

Quantifiers

Quantifiers are used to check how many times a character should occur.
Some commonly used quantifiers are-

1. *

It matches zero or more occurrences of any given character.

```
JavaScript
const regex = /go*d/;
console.log(regex.test("gd"));
console.log(regex.test("god"));
console.log(regex.test("good"));
console.log(regex.test("g"));

// output
// true
// true
// true
// false
```

In the above example, we can see that in first console there is zero ‘o’ character after character ‘g’, in second there is one ‘o’ character after character ‘g’, in third there is two ‘o’ character after character ‘g’ that's why they are returning true, but in the last console, we had removed character ‘d’ which is a part of our pattern ending that's why we are getting false over there because as per our regex we can have any number of occurrences of character ‘o’ in between character ‘g’ and ‘d’.

2. +

It matches for the one or more occurrences of any given character.

```
JavaScript
const regex = /go+d/;
console.log(regex.test("gd"));
console.log(regex.test("god"));
console.log(regex.test("good"));
console.log(regex.test("g"));

// output
// false
// true
// true
// false
```

In the above example, we can see that the first console is false as there is no ‘o’ character in between character ‘g’ and ‘d’, but in case of second and third log we have one and two character, that's why getting true over there and again in last getting false as it has missing ending character of our regex.

3. ?

It matches for the zero or one occurrences only of any given character.

```
JavaScript
const regex = /go?d/;
console.log(regex.test("gd"));
console.log(regex.test("god"));
console.log(regex.test("good"));
console.log(regex.test("g"));

// output
// true
// true
// false
// false
```

In the above example, we are getting true for the first and second console as it has zero and one occurrence only of the character ‘o’ in it and getting false for rest as the third one has two occurrences and fourth one has missing ending character.

4. {n}

It matches for the exact number of occurrences only of any given character. If less or more than that then will return false or null in case of match method.

```
JavaScript
const regex = /go{2}d/;
console.log(regex.test("gd"));
console.log(regex.test("god"));
console.log(regex.test("good"));
console.log(regex.test("goood"));

// output
// false
// false
// true
// false
```

In the above example, except the third console all are returning false as they do not have exactly 2 occurrences of the given character in it. Either they have less or more but not the exact occurrences.

5. {n,}

With this modifier, we can have n or more than n number of occurrences.

```
JavaScript
const regex = /go{2,}d/;
console.log(regex.test("gd"));
console.log(regex.test("god"));
console.log(regex.test("good"));
console.log(regex.test("goood"));

// output
// false
// false
// true
// true
```

In the above example, we are getting true for the last two only as they have 'n' or more than 'n' number of occurrences.

Anchors and Boundaries

Anchors and boundaries are used to specify where the match should occur within the string.
Some commonly used anchors and boundaries are-

1. ^

This is used to check matches at the start of the string.

```
JavaScript
const regex = /^PW/;
console.log(regex.test("PW Skills"));
console.log(regex.test("iNeuron"));

// output
// true
// false
```

As we can see the output that we are getting is true and false as the first string starts with the given regex pattern but the second one is not.

2. $

This is used to check matches at the end of the string.

```javascript
const regex = /s$/;
console.log(regex.test("PW Skills"));
console.log(regex.test("iNeuron"));

// output
// true
// false
```

In the above example, we can see that we are getting true for the first log as the string is ending with the given character in regex but the second one is false as it is not ending with the given pattern in regex.

Note
'^' is always placed in the beginning of the regex character to which we want to match but '$' will be after the character to which we want to match.

3. \b

It is used to match the pattern only if it appears at a word boundary, meaning it should be surrounded by word characters (letters, digits, or underscores). It would not match if the pattern is a part of a larger word.

```javascript
const regex = /\bSkills\b/;
console.log(regex.test("PW Skills"));
console.log(regex.test("iNeuron"));

// output
// true
// false
```

In the above example, we are getting true for first as the given words in pattern are over there but not in the second one that's why we are getting false on the second console.

Grouping and Capturing
In javascript regex, we can use parentheses to group and capture parts of a pattern. By capturing a group, it allows us to extract specific portions of the matched text.

Grouping
Grouping is used to group a part of a regular expression together. It is quite helpful when we want to apply quantifiers or other operators to a specific part of the pattern.

```javascript
console.log(/(apple|banana)/.test("This is an apple"));
console.log(/(apple|banana)/.test("This is a banana"));
console.log(/(apple|banana)/.test("This is an orange"));

// Output
// true
// true
// false
```

In the above example, we are using the pipe symbol (|) to check for or condition something like logical or which we had discussed while learning more about operators. So the pipe symbol allows us to check for this or that so if any of the values will match the pattern it will return true.
We had wrapped our pattern inside the parenthesis to form a group of it. As in the output we can see that we had true for the first two cases as in first apple is there and in second banana is there but in third none of them is there that's why we are getting false over there.

Note: We will see more complex examples which will use grouping in further classes.

Capturing
Capturing groups allows us to extract the matched portions of the text. To create a capturing group, we can enclose the pattern we want to capture within parenthesis.

```javascript
const email1 = "pw@gmail.com";
const email2 = "pw@google.test";
const email3 = "pw";
console.log(email1.match(/@(\w+\.\w+)/));
console.log(email2.match(/@(\w+\.\w+)/));
console.log(email3.match(/@(\w+\.\w+)/));

// Output
// [
//   '@gmail.com',
//   'gmail.com',
//   index: 2,
//   input: 'pw@gmail.com',
//   groups: undefined
// ]
```

```
// [
//   '@google.test',
//   'google.test',
//   index: 2,
//   input: 'pw@google.test',
//   groups: undefined
// ]
// null
```

Before diving into the output let's explore the regex code first and breakdown its pattern-

1. '@'

This is used to match the '@' symbol, which is required in an email.

2. '(\w+\.\w+)'

This is a capturing group, enclosed within the parenthesis. We are using 'w+' which means it will match for one or more word characters (letters, digits, or underscores), this is making sure that at least one character will be there before and after the dot. Also we are using '\.\' to check for the dot character in our email id.

3. As we can see in the output that for the first two we are able to extract the domain name of our email as it is there in the string but getting null for the third one as there is no such domain available.

Lookahead and Look behind

Lookaheads (?=..) and Lookbehind (?<=..) allows us to assert that a pattern should or should not be followed or preceded by another pattern, without including it in the match.

Example of lookahead

JavaScript

```
// Example of lookahead
console.log(/pw(?=skills)/.test("pw skills"));
console.log(/pw(?= skills)/.test("pw skills"));

// output
// false
// true
```

In the above example of lookahead, it is checking that the pw is followed by skills or not. We can see that the first console is logging false as after equal to symbol in regex there is no space but getting true in the second one as it has space in it also.

Example of lookbehind

```javascript
// Example of lookbehind
console.log(/(?<=pw)skills/.test("pw skills"));
console.log(/(?<=pw) skills/.test("pw skills"));

// output
// false
// true
```

Similar to the above example, we are checking that the 'skills' is preceded by pw or not so getting false for first as it has no space but getting true after matching space also.

Some common use cases

1. **Validating the email addresses**

```javascript
const email1 = "pw@skills.com";
const email2 = "pwskills.in";
const regex = /^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$/;
console.log(regex.test(email1));
console.log(regex.test(email2));

// output
// true
// false
```

First let's understand the above used regex code-

1. ^[a-zA-Z0-9._%+-]+

It checks that the email should start from uppercase, lowercase,number or with the above symbols.

2. @

It checks for the '@' symbol in the pattern.

3. [a-zA-Z0-9.-]+Similar to the first one after the '@' symbol, there can be uppercase, lowercase, number or dash symbols.

4. .[a-zA-Z]{2,}$

This one is checking for the dot symbol and after that there should be an uppercase or lowercase character which should be of at least two or more characters.

5. That is the reason we are getting true for email1 and false for email2 as per our regex check.

Validating the indian phone number

```javascript
const regex = /^[987][0-9]{9}/;
console.log(regex.test("9874563218"));
console.log(regex.test("5874563218"));
console.log(regex.test("787456321"));
console.log(regex.test("787456321a"));

// output
// true
// false
// false
// false
```

Let's understand the used regex-

1. ^[987]

It is checking that the number should always start from the 9,8 or 7 always.

2. [0-9]{9}

It is for assuring that there should be 9 more characters which can be a number between 0 to 9.

3. As per the regex we are getting those outputs.