JS

JavaScript Fundamentals

By:Ahmed ElMahdy

Date and Time

Using the Date Object

- The Date object is a built-in JavaScript object.
- It allows you to get the user's local time by accessing the computer system clock through the browser.
- The Date object also provides several methods for managing, manipulating, and formatting dates and times.

Creating a Date Object

Before we start working with the date and time, we need to create a Date object. Unlike other built-in objects, such as arrays or functions, dates don't have a corresponding literal form: all date objects need to be created using the Date constructor function which is Date().
There are four different ways to create a Date object in JavaScript.

The new Date() **Syntax**

- Declare a new Date object without initializing its value.
- The date and time value will be set to the current date and time on the user's device on which the script is run.

The new Date(year, month, ...) Syntax You can also initialize a Date object by passing the following parameters separated by commas: year, month, day, hours, minutes, seconds, and milliseconds. The year and month parameters

Date and Time

Creating a Date Object

The new Date(*year, month,* ...) Syntax

- Initialize a Date object by passing the following parameters separated by commas: year, month, day, hours, minutes, seconds, and milliseconds.
- The year and month parameters are required other parameters are optional
- You can ignore the time part and specify just the date part if you wish.

The new Date(*dateString*) Syntax

- JavaScript also allows you to create a Date object by passing the string representing a date, or a date and time.

The new Date(*milliseconds*) Syntax

- You can also define a Date object by passing the number of milliseconds since January 1, 1970, at 00:00:00 GMT.
- This time is known as the UNIX epoch because 1970 was the year when the UNIX operating system was formally introduced.

Once you have created an instance of the Date object, you can use its methods to perform various tasks, such as getting different component of a date, setting or modifying individual date and time value, etc.

Date and Time

Getting the Current Date and Time

- Create a new Date object without passing any parameters.
- This will create an object with the current date and time.

Creating the Date and Time Strings

- The JavaScript Date object provides several methods, such as toDateString(), toLocaleDateString(), etc. to generate date strings in different formats.
- you can use the toLocaleTimeString(), toTimeString() methods of the Date object to generate time strings.

Getting Specific Date and Time Components

Once you have a proper date object, a number of methods are available to you to extract details from it, such as the month, date, hours or minutes value etc.
The following section describes the various methods of extracting individual pieces of information from a Date object.

Getting the Year, Month and Date

The Date object provides several methods such as getFullYear(), getMonth(), getDay(), etc. that you can use to extract the specific date components from the Date object, such as year, day of month, day of week, etc.

Date and Time

Getting Specific Date and Time Components

Getting the Hours, Minutes, Seconds, and Milliseconds

- The Date object provides methods like getHours(), getMinutes(), getSeconds(), getTimezoneOffset() etc. to extract the time components from the Date object.
- The getHours() method returns the number of hours into the day (from 0 to 23) according to the 24-hour clock. So, when it is midnight, the method returns 0; and when it is 3:00 P.M., it returns 15.

Note: The Date objects also have methods to obtain the UTC components. Just place UTC after get, such as getUTCDate(), getUTCHour(), getUTCMinutes(), and so on.

Date and Time

Setting the Date and Time Values

In addition to retrieving date and time values, you can also set or modify these values using the JavaScript. This is most often used in program where you have to change the value of a date object from one particular date or time to another.

Let's see how it works.

Setting the Year, Month and Date

- The Date object provides methods such as setFullYear(), setMonth() and setDate() methods to set the year, month, date components of the Date object respectively.
- You can use the setMonth() method to set or modify the month part of a Date object.
- The setMonth() method require an integer value from 0 to 11, if you set the value of the month greater than 11, the year value of the date object will increment.
- The setDate() method require an integer value from 1 to 31. Also, if you pass the values greater than the number of days in the month, the month will increment.

Date and Time

Setting the Date and Time Values

Setting the Hours, Minutes and Seconds

- Methods for setting the time values are also pretty straight forward.
- The setHours(), setMinutes(), setSeconds(), setMilliseconds() can be used to set the hour, minutes, seconds, and milliseconds part of the Date object respectively.
- Each method takes integer values as parameters.
- Hours range from 0 to 23. Minutes and seconds range from 0 to 59. And milliseconds range from 0 to 999.

Event Listeners

Understanding Event Listeners

The event listeners are just like event handlers, except that you can assign as many event listeners as you like to a particular event on particular element.

Adding Event Listeners for Different Event Types

Like event handler, you can assign different event listeners to different event types on the same element.

Adding Event Listeners to Window Object

The addEventListener() method allows you to add event listeners to any HTML DOM elements, the document object, the window object, or any other object that support events, e.g, XMLHttpRequest object.

Removing Event Listeners

You can use the removeEventListener() method to remove an event listener that have been previously attached with the addEventListener().

Note: The addEventListener() and removeEventListener() methods supported in all major browsers. Not supported in IE 8 and earlier, and Opera 6.0 and earlier versions.

Event Propagation

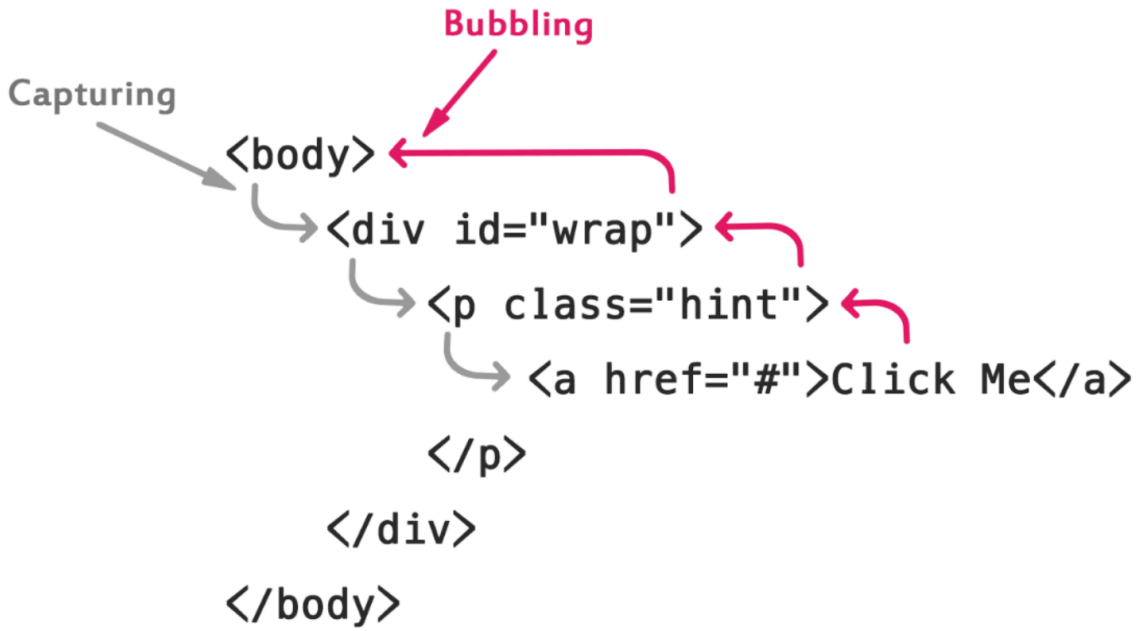
Understanding the Event Propagation
Event propagation is a mechanism that defines how events propagate or travel through the DOM tree to arrive at its target and what happens to it afterward.

Let's understand this with the help of an example, suppose you have assigned a click event handler on a hyperlink (i.e. element) which is nested inside a paragraph (i.e. element). Now if you click on that link, the handler will be executed. But, instead of link, if you assign the click event handler to the paragraph containing the link, then even in this case, clicking the link will still trigger the handler. That's because events don't just affect the target element that generated the event—they travel up and down through the DOM tree to reach their target.
This is known as event propagation

In modern browser event propagation proceeds in two phases: capturing, and bubbling phase. Before we proceed further, take a look at the following illustration:

Event Propagation

Understanding the Event Propagation



Above image demonstrates how event travels in the DOM tree during different phases of the event propagation when an event is fired on an element that has parent elements.

Event Propagation

Understanding the Event Propagation

The concept of event propagation was introduced to deal with the situations in which multiple elements in the DOM hierarchy with a parentchild relationship have event handlers for the same event, such as a mouse click. Now, the question is which element's click event will be handled first when the user clicks on the inner element—the click event of the outer element, or the inner element.

In the following sections of this chapter we will discuss each phases of the event propagation in greater detail and find out the answer of this question.

Note: Formally there are 3 phases, capture, target and bubble phase. But, the 2nd phase i.e. the target phase (occurs when the event arrives at the target element that has generated the event) is not handled separately in modern browsers, handlers registered for both *capturing* and *bubbling* phases are executed in this phase.

Event Propagation

Event Propagation phase

The Capturing Phase

- Events propagate from the Window down through the DOM tree to the target node. For example, if the user clicks a hyperlink, that click event would pass through the <html> element, the <body> element, and the <p> element containing the link.
- If any ancestor (i.e. parent, grandparent, etc.) of the target element and the target itself has a specially registered **capturing event listener** for that type of event, those listeners are executed during this phase.
- Event capturing is not supported in all browsers and rarely used. For instance, Internet Explorer prior to version 9.0 does not support event capturing.
- Event capturing only works with event handlers registered with the addEventListener() method when the third argument is set to true.
- The traditional method of assigning event handlers, like using onclick, onmouseover, etc. won't work here.

Event Propagation

Event Propagation phase

The Bubbling Phase

- In the bubbling phase, the exact opposite occurs.
- In this phase event propagates or bubbles back up the DOM tree, from the target element up to the Window, visiting all of the ancestors of the target element one by one.
- Example, if the user clicks a hyperlink, that click event would pass through the <p> element containing the link, the <body> element, the <html> element, and the document node.
- Also, if any ancestor of the target element and the target itself has event handlers assigned for that type of event, those handlers are executed during this phase.
- In modern browsers, all event handlers are registered in the bubbling phase, by default.

Event Propagation

Accessing the Target Element

- The target element is the DOM node that has generated the event.
- Example, if the user clicks a hyperlink, the target element is the hyperlink.
- The target element is accessible as event.target, it doesn't change through the event propagation phases.
- The this keyword represents the current element (i.e. the element that has a currently running handler attached to it).

Stopping the Event Propagation

- You can also stop event propagation in the middle if you want to prevent any ancestor element's event handlers from being notified about the event.
- Example, suppose you have nested elements and each element has onclick event handler that displays an alert dialog box.
- When you click on the inner element all handlers will be executed at once, since event bubble up to the DOM tree.
- You can even prevent any other listeners attached to the same element for the same event type from being executed using the stopImmediatePropagation() method.

Note: If several listeners are attached to the same element for the same event type, they are executed in order in which they have been added. But, if any listener calls the event.stopImmediatePropagation() method, no remaining listeners will be executed..

Event Propagation

Preventing the Default Action

- Some events have a default action associated with them.
- Example, if you click on a link browser takes you to the link's target, when you click on a form submit button browser submit the form, etc.
- You can prevent such default actions with the preventDefault() method of the event object.

Borrowing Methods

In JavaScript, you can borrow methods from other objects to build some functionality without inheriting all their properties and methods. JavaScript provides two methods for all function objects, call() and apply(), that allow a function to be invoked as if it were a method of another object.

Difference Between call() **and** apply() **Methods**

The syntax of the apply() method is almost identical to call().

The only difference is, the call() method takes a list of arguments like call(*thisObj, arg1, arg2, ...*).

While the apply() method takes a single array of arguments like apply(*thisObj,* [*argsArray*])

Using Built-in Methods

- The apply() method also allows you to use built-in methods for performing some tasks quickly and easily.
- One such example is using the Math.max()/Math.min() to find out the maximum or minimum value in an array, that would otherwise require looping over the array values.
- JavaScript arrays do not have a max() method, but Math has, so we can apply the Math.max() method.

Note: The first argument to both call() and apply() is the object on which the function is to be invoked. Using null as the first argument is like calling the function without providing any object for the this pointer inside the function.

Borrowing Methods

Using Built-in Methods

- The new ES6 spread operator provides a shorter way to obtain the maximum or minimum value from an array without using the apply() method.
- Both spread (...) and apply() will either fail or return the incorrect result if the array has too many elements (e.g. tens of thousands). In that case you can use the Array.reduce() to find the maximum or minimum value in a numeric array, by comparing each value.

Hoisting

What is Hoisting

In JavaScript, all variable and function declarations are moved or hoisted to the top of their current scope, regardless of where it is defined. This is the default behavior of JavaScript interpreter which is called hoisting.
In the following sections we'll take a closer look at how it actually works.

Function Hoisting

Functions that are defined using a function declaration are automatically hoisted. That means they can be called before they have been defined.

Variable Hoisting

- The variable declarations are also hoisted to the top of their current scope automatically.
- If the variable is declared inside a function block, it will be moved at the top of the function.
- If it is declared outside any function it will be moved to top of the script and become globally available.
- JavaScript only hoists declarations, not initializations. That means if a variable is declared and initialized after using it, the value will be undefined.

Note: It is considered best practice to declare your variables at the top of the current scope, because of hoisting behavior. Also, using a variable without declaring is not allowed in JavaScript strict mode.

Closures

Understanding the JavaScript Closures

- In the JavaScript functions chapter you've learnt that in JavaScript a variable's scope can be *global* or *local*.
- Since ES6 you can also create blockscoped variables using the let keyword.
- A global variable can be accessed and manipulated anywhere in the program, whereas a local variable can only be accessed and manipulated by the function they are declared in.
- There are certain situations when you want a variable to be available throughout the script, but you don't want just any part of your code to be able to change its value accidently.
- A closure is basically an inner function that has access to the parent function's scope, even after the parent function has finished executing. This is accomplished by creating a function inside another function.

Tip: In JavaScript, all functions have access to the global scope, as well as the scope above them. As JavaScript supports nested functions, this typically means that the nested functions have access to any value declared in a higher scope including its parent function's scope.

Note: The global variables live as long as your application (i.e. your web page) lives. Whereas, the local variables have a short life span, they are created when the function is invoked, and destroyed as soon as the function is finished executing.

Closures

Creating the Getter and Setter Functions

- Here we will create a variable secret and protect it from being directly manipulated from outside code using closure.
- We will also create getter and setter functions to get and set its value.
- The setter function will also perform a quick check whether the specified value is a number or not, and if it is not it will not change the variable value.

Tip: Self-executing functions are also called immediately invoked function expression (IIFE), immediately executed function, or self-executing anonymous function.

Strict Mode

General Restrictions in Strict Mode
Strict mode changes both syntax and runtime behaviour.

Undeclared Variables are Not Allowed
In strict mode, all variables must be declared. if you assign a value to an identifier that is not a declared variable, a ReferenceError will be thrown.

Deleting a Variable or a Function is Not Allowed
In strict mode, if you try to delete a variable or a function, a syntax error will be thrown. Whereas, in non-strict mode, such attempt fails silently and the delete expression evaluates to false.

Duplicating a Parameter Name is Not Allowed
In strict mode, a syntax error will be thrown, if a function declaration has two or more parameters with the same name. In non-strict mode, no error occurs.

Strict Mode

The eval Method Cannot Alter Scope

In strict mode, for security reasons, code passed to eval() cannot declare/modify variables or define functions in the surrounding scope as it can in non-strict mode.

The eval and arguments Cannot be Used as Identifiers

In strict mode, the names eval and arguments are treated like keywords, so they cannot be used as variable names, function names, or as function parameter names, etc.

The with Statement is Not Allowed

- In strict mode, the with statement is not allowed.
- The with statement adds the properties and methods of the object to the current scope.
- The statements nested inside the with statement can call the properties and methods of the object directly without referring it.

Writing to a Read-only Property is Not Allowed

In strict mode, assigning value to a non-writable property, a get-only property or a non-existing property will throw an error. In non-strict mode, these attempts fail silently.

Strict Mode

Adding a New Property to a Non-extensible Object is Not Allowed

In strict mode, attempts to create new properties on non-extensible or non-existing objects will also throw an error. But in non-strict mode, these attempts fail silently.

Octal Numbers are Not Allowed

- In strict mode, octal numbers (numbers prefixed with a zero e.g. 010, 0377) are not allowed. Though, it is supported in all browsers in non-strict mode.
- However, in ES6 octal numbers are supported by prefixing a number with 0o i.e. 0o10, 0o377, etc.

Keywords Reserved for Future are Not Allowed

- The reserved words cannot be used as identifier (variable names, function names, and loop labels) in a JavaScript program.
- The strict mode also imposes restrictions on uses of those keywords that are reserved for future.
- As per the latest ECMAScript 6 (or ES6) standards, these keywords are reserved keywords when they are found in strict mode code: await, implements, interface, package, private, protected, public, and static.
- For optimal compatibility you should avoid using the reserved keywords as variable names or function names in your program

JSON Parsing

What is JSON

JSON stands for JavaScript Object Notation. JSON is extremely lightweight data-interchange format for data exchange between server and client which is quick and easy to parse and generate.

Like XML, JSON is also a text-based format that's easy to write and easy to understand for both humans and computers, but unlike XML, JSON data structures occupy less bandwidth than their XML versions.

JSON is based on two basic structures:

• Object: This is defined as an unordered collection of key/value pairs (i.e. key:value). Each object begins with a left curly bracket { and ends with a right curly bracket }. Multiple key/value pairs are separated by a comma ,.

• Array: This is defined as an ordered list of values. An array begins with a left bracket [and ends with a right bracket]. Values are separated by a comma ,.

In JSON, property names or keys are always strings, while the value can be a string, number, true or false, null or even an object or an array. Strings must be enclosed in double quotes " and can contain escape characters such as \n, \t

Tip: A data-interchange format is a text format which is used to interchange or exchange data between different platforms and operating systems. JSON is the most popular and lightweight data-interchange format for web applications.

JSON Parsing

Parsing JSON Data in JavaScript

- Parse JSON data received from the web server using the JSON.parse() method.
- This method parses a JSON string and constructs the JavaScript value or object described by the string.
- If the given string is not valid JSON, you will get a syntax error.

Parsing Nested JSON Data in JavaScript

- JSON objects and arrays can also be nested.
- A JSON object can arbitrarily contains other JSON objects, arrays, nested arrays, arrays of JSON objects, and so on.

Encoding Data as JSON in JavaScript

Sometimes JavaScript object or value from your code need to be transferred to the server during an Ajax communication. JavaScript provides JSON.stringify() method for this purpose which converts a JavaScript value to a JSON string.

Stringify a JavaScript Object

Stringify a JavaScript Array

Error Handling

Handling Errors

Sometimes your JavaScript code does not run as smooth as expected, resulting in an error. There are a number of reasons that may cause errors, for instance:

- A problem with network connection
- A user might have entered an invalid value in a form field
- Referncing objects or functions that do not exist
- Incorrect data being sent to or received from the web server
- A service that the application needs to access might be temporarily unavailable

These types of errors are known as runtime errors, because they occur at the time the script runs. A professional application must have the capabilities to handle such runtime error gracefully. Usually this means informing the user about the problem more clearly and precisely.

Error Handling

The try...catch Statement

- JavaScript provides the try-catch statement to trap the runtime errors, and handle them gracefully.
- Any code that might possibly throw an error should be placed in the try block of the statement, and the code to handle the error is placed in the catch block.
- If an error occurs at any point in the try block, code execution immediately transferred from the try block to the catch block.
- If no error occurs in the try block, the catch block will be ignored, and the program will continue executing after the try-catch statement.

Note: the catch keyword is followed by an identifier in parentheses. This identifier is act like a function parameter. When an error occurs, the JavaScript interpreter generates an object containing the details about it. This error object is then passed as an argument to catch for handling.

Tip: The try-catch statement is an exception handling mechanism. An exception is signal that indicates that some sort of exceptional condition or error has occurred during the execution of a program. The terms "exception" and "error" are often used interchangeably.

Error Handling

The try...catch...finally Statement

The try-catch statement can also have a finally clause. The code inside the finally block will always execute, regardless of whether an error has occurred in the try block or not.

Throwing Errors

- So far we've seen the errors that are automatically thrown by JavaScript parser when an error occurs. However, it is also possible to throw an error manually by using the throw statement.
- The general form (or syntax) of the throw statement is: throw *expression*;
- The expression can be a object or a value of any data type.
- Use the objects, preferably with name and message properties.
- The JavaScript built-in Error() constructor provides a convenient way create an error object.

Note: If you're using the JavaScript built-in error constructor functions (e.g. Error(), TypeError(), etc.) for creating error objects, then the name property is same as the name of the constructor, and the message is equal to the argument passed to the constructor function.

Tip: Theoretically it is possible to calculate the square root of negative number by using the imaginary number i, where i 2 = -1. Therefore square root of -4 is 2i, square root of -9 is 3i, and so on. But imaginary numbers are not supported in JavaScript.

Error Handling

Error Types

- The Error object is the base type of all errors and it has two main properties — a name property that specifies the *type of error*, and a message property that holds a message describing the error in more detail. Any error thrown will be an instance of the Error object.
- There are several different types of error that can occur during the execution of a JavaScript program, such as RangeError, ReferenceError, SyntaxError, TypeError, and URIError.

The following section describes each one of these error type in more detail:

RangeError

A RangeError is thrown when you use a number that is outside the range of allowable values. For example, creating an array with a negative length will throw RangeError.

ReferenceError

A ReferenceError is typically thrown when you try to reference or access a variable or object that doesn't exist. The following example shows how the ReferenceError occurs.

Error Handling

Error Types

SyntaxError

A SyntaxError is thrown at runtime if there is any syntax problem in your JavaScript code. For example, if closing bracket is missing, loops are not structured properly, and so on.

TypeError

A TypeError is thrown when a value is not of the expected type. For example, calling a string method on number, calling an array method on string, and so on.

URIError

A URIError is thrown when you specified an invalid URI (*stands for* Uniform Resource Identifier) to the URI-related functions such as encodeURI() or decodeURI().

Note: There is one more error type EvalError which is thrown when an error occurs during the execution of code via eval() function. But, this error is not thrown by JavaScript anymore, however this object still remains for backward compatibility

Error Handling

Error Types

The specific error type can also be thrown manually using their respective constructor and the throw statement, e.g., to throw a TypeError you can use the TypeError() constructor.

Note: The Error object also supports some non-standard properties. One of the most widely used such property is: stack, which returns the stack trace for that error. You can use it for debugging purposes, but don't use it on production sites.

Regular Expressions

What is Regular Expression

Regular Expressions, commonly known as "**regex**" or "**RegExp**", are a specially formatted text strings used to find patterns in text.

Regular expressions are one of the most powerful tools available today for effective and efficient text processing and manipulations. For example, it can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user is correct or not, find or replace matching string within text content, and so on.

JavaScript supports Perl style regular expressions.

Why Perl style regular expressions? Because Perl (Practical Extraction and Report Language) was the first mainstream programming language that provided integrated support for regular expressions and it is well known for its strong support of regular expressions and its extraordinary text processing and manipulation capabilities.

Let's begin with a brief overview of the commonly used JavaScript's built-in methods for performing pattern-matching before delving deep into the world of regular expressions.

Regular Expressions

What is Regular Expression

Function	What it Does
`exec()`	Search for a match in a string. It returns an array of information or `null` on mismatch.
`test()`	Test whether a string matches a pattern. It returns `true` or `false`.
`search()`	Search for a match within a string. It returns the index of the first match, or `-1` if not found.
`replace()`	Search for a match in a string, and replaces the matched substring with a replacement string.
`match()`	Search for a match in a string. It returns an array of information or `null` on mismatch.
`split()`	Splits up a string into an array of substrings using a regular expression.

Note: The methods exec() and test() are RegExp methods that takes a string as a parameter, whereas the methods search(), replace(), match() and split() are String methods that takes a regular expression as a parameter.

Regular Expressions

Defining Regular Expressions

- In JavaScript, regular expressions are represented by RegExp object, which is a native JavaScript object like String, Array, and so on.
- There are two ways of creating a new RegExp object — one is using the literal syntax, and the other is using the RegExp() constructor.
- The literal syntax uses forward slashes *(/pattern/)* to wrap the regular expression pattern, whereas the constructor syntax uses quotes ("pattern").

Note: When using the constructor syntax, you've to double-escape special characters, which means to match "." you need to write "\\." instead of "\.". If there is only one backslash, it would be interpreted by JavaScript's string parser as an escaping character and removed.

Regular Expressions

Pattern Matching with Regular Expression

Regular expression patterns include the use of letters, digits, punctuation marks, etc., plus a set of special regular expression characters (do not confuse with the HTML special characters).

The characters that are given special meaning within a regular expression, are:
. * ? + [ ] ( ) { } ^ $ | \. You will need to backslash these characters whenever you want to use them literally.

For example, if you want to match ".", you'd have to write \.. All other characters automatically assume their literal meanings.

The following sections describe the various options available for formulating patterns:

Regular Expressions

Pattern Matching with Regular Expression

Character Classes

- Square brackets surrounding a pattern of characters are called a character class e.g. [abc].
- A character class always matches a single character out of a list of specified characters that means the expression [abc] matches only a, b or c character.
- Negated character classes can also be defined that match any character except those contained within the brackets.
- A negated character class is defined by placing a caret (^) symbol immediately after the opening bracket, like [^abc], which matches any character except a, b, and c.
- Define a range of characters by using the hyphen (-) character inside a character class, like [0-9]. Let's look at some examples of the character classes:

RegExp	What it Does
[abc]	Matches any one of the characters a, b, or c.
[^abc]	Matches any one character other than a, b, or c.
[a-z]	Matches any one character from lowercase a to lowercase z.
[A-Z]	Matches any one character from uppercase a to uppercase z.
[a-Z]	Matches any one character from lowercase a to uppercase Z.
[0-9]	Matches a single digit between 0 and 9.
[a-z0-9]	Matches a single character between a and z or between 0 and 9.

Regular Expressions

Pattern Matching with Regular Expression

Predefined Character Classes

Some character classes such as digits, letters, and whitespaces are used so frequently that there are shortcut names for them. The following table lists those predefined character classes:

Shortcut	What it Does
.	Matches any single character except newline `\n`.
`\d`	matches any digit character. Same as `[0-9]`
`\D`	Matches any non-digit character. Same as `[^0-9]`
`\s`	Matches any whitespace character (space, tab, newline or carriage return character). Same as `[ \t\n\r]`

Shortcut	What it Does
`\S`	Matches any non-whitespace character. Same as `[^ \t\n\r]`
`\w`	Matches any word character (definned as a to z, A to Z,0 to 9, and the underscore). Same as `[a-zA-Z_0-9]`
`\W`	Matches any non-word character. Same as `[^a-zA-Z_0-9]`

Regular Expressions

Pattern Matching with Regular Expression

Repetition Quantifiers

- what if you want to match on more than one character? For example, let's say you want to find out words containing one or more instances of the letter p, or words containing at least two p's, and so on.
- This is where quantifiers come into play.
- With quantifiers you can specify how many times a character in a regular expression should match.
- Quantifiers can be applied to the individual characters, as well as classes of characters, and groups of characters contained by the parentheses.

The following table lists the various quantifiers for the pattern:

RegExp	What it Does
`p+`	Matches one or more occurrences of the letter p.
`p*`	Matches zero or more occurrences of the letter p.
`p?`	Matches zero or one occurrences of the letter p.
`p{2}`	Matches exactly two occurrences of the letter p.
`p{2,3}`	Matches at least two occurrences of the letter p, but not more than three occurrences.
`p{2,}`	Matches two or more occurrences of the letter p.
`p{,3}`	Matches at most three occurrences of the letter p

Regular Expressions

Pattern Matching with Regular Expression

Position Anchors

- There are certain situations where you want to match at the beginning or end of a line, word, or string.
- To do this you can use anchors.
- Two common anchors are caret (^) which represent the start of the string, and the dollar ($) sign which represent the end of the string.

RegExp	What it Does
^p	Matches the letter p at the beginning of a line.
p$	Matches the letter p at the end of a line.

Regular Expressions

Pattern Matching with Regular Expression

Pattern Modifiers (Flags)

- A pattern modifier allows you to control the way a pattern match is handled.
- Pattern modifiers are placed directly after the regular expression, for example, if you want to search for a pattern in a case-insensitive manner, you can use the i modifier, like this: /pattern/i.

The following table lists some of the most commonly used pattern modifiers.

Modifier	What it Does
`g`	Perform a global match i.e. finds all occurrences.
`i`	Makes the match case-insensitive manner.
`m`	Changes the behavior of `^` and `$` to match against a newline boundary (i.e. start or end of each line v multiline string), instead of a string boundary.
`o`	Evaluates the expression only once.
`s`	Changes the behavior of `.` (dot) to match all characters, including newlines.
`x`	Allows you to use whitespace and comments within a regular expression for clarity.

Regular Expressions

Pattern Matching with Regular Expression

Alternation

Alternation allows you to specify alternative version of a pattern.

Alternation in a regular expression works just like the OR operator in an ifelse conditional statement.

You can specify alternation using a vertical bar (|). For example, the regexp /fox|dog|cat/ matches the string "fox", or the string "dog", or the string "cat".

Note: Alternatives are evaluated from left to right until a match is found. If the left alternative matches, the right alternative is ignored completely even if it has a match.

Word Boundaries

A word boundary character (\b) helps you search for the words that begins and/or ends with a pattern.

Regular Expressions

Pattern Matching with Regular Expression

Grouping

- Regular expressions use parentheses to group subexpressions, just like mathematical expressions.
- Parentheses allow a repetition quantifier to be applied to an entire subexpression.
- For example, in regexp /go+/ the quantifier + is applied only to the last character o and it matches the strings "go", "goo", and so on. Whereas, in regexp /(go)+/ the quantifier + is applied to the group of characters g and o and it matches the strings "go", "gogo", and so on.

Note: If the string matches the pattern, the match() method returns an array containing the entire matched string as the first element, followed by any results captured in parentheses, and the index of the whole match. If no matches were found, it returns null.

Tip: If the regular expression includes the g flag, the match() method only returns an array containing all matched substrings rather than match object. Captured groups, index of the whole match, and other properties are not returned.

Form Validation

Understanding Client-Side Validation

Web forms have become an essential part of web applications. It is often used to collect user's information such as name, email address, location, age, and so on. But it is quite possible that some user might not enter the data what you've expected. So to save bandwidth and avoid unnecessary strain on your server resources you can validate the form data on clientside (i.e. user's system) using JavaScript before passing it onto the web server for further processing.

Client-side validation is also helpful in creating better user experience, since it is faster because validation occurs within the user's web browser, whereas server-side validation occurs on the server, which require user's input to be first submitted and sent to the server before validation occurs, also user has to wait for server response to know what exactly went wrong.

In the following section we will take a closer look at how to perform JavaScript form validation and handle any input errors found appropriately and gracefully.

Note: Client-side validation is not a substitute or alternative for server-side validation. You should always validate form data on the server-side even if they are already validated on the client-side, because user can disable JavaScript in their browser.

Form Validation

The form validation process typically consists of two parts— the required fields validation which is performed to make sure that all the mandatory fields are filled in, and the data format validation which is performed to ensure that the type and format of the data entered in the form is valid.
Well, let's get straight to it and see how this actually works.

Creating the HTML Form
Let's first create a simple HTML form that we will validate on client-side using JavaScript when the user clicks on the submit button.

Building the Form Validation Script
Now we're going to create a JavaScript file that holds our complete validation script.

Adding Style Sheet to Beautify the Form