WHAT IS JSON?

- JSON stands for **J**ava**S**cript **O**bject **N**otation
- JSON is a lightweight text-based, designed explicitly for human-readable data interchange.
- It is a language-independent data format. It supports almost every kind of language, framework, and library.
- JSON is "self-describing" and easy to understand

JavaScript Object Notation (full form of JSON) is a standard file format used to interchange data. The data objects are stored and transmitted using key-value pairs and array data types. In simpler terms, JSON data is (in some ways) the language of databases.

JSON has syntactic similarity to JavaScript because JSON is based on JavaScript object notation syntaxes. But JSON format is text only, which makes it easy to read and use with any programming language.

WHY DO WE USE JSON?

Since JSON is an easy-to-use, lightweight language data interchange format in comparison to other available options, it can be used for API integration. Following are the advantages of JSON:

- **Less Verbose**: In contrast to XML, JSON follows a compact style to improve its users' readability. While working with a complex system, JSON tends to make substantial enhancements.
- **Faster**: The JSON parsing process is faster than that of the XML because the DOM manipulation library in XML requires extra memory for handling large XML files. However, JSON requires less data that ultimately results in reducing the cost and increasing the parsing speed.
- **Readable**: The JSON structure is easily readable and straightforward. Regardless of the programming language that you are using, you can easily map the domain objects.

- **Structured Data**: In JSON, a map data structure is used, whereas XML follows a tree structure. The key-value pairs limit the task but facilitate the predictive and easily understandable model.

JSON became so popular that it is used for data for all kinds of applications. It is the most popular way of sending the data for Web APIs.

The JSON format is syntactically similar to the code for creating JavaScript objects. Because of this, a JavaScript program can easily convert JSON data into JavaScript objects.

Since the format is text only, JSON data can easily be sent between computers, and used by any programming language.

JavaScript has a built in function for **converting JSON strings into JavaScript objects:**

JSON.parse()

JavaScript also has a built in function for converting an object into a JSON string:

JSON.stringify()

JSON Vs XML

JSON stands for JavaScript Object Notation, whereas XML stands for Extensive Markup Language. Nowadays, JSON and XML are widely used as data interchange formats, and both have been acquired by applications as a technique to store structured data.

Difference between JSON and XML:

JSON is easy to learn.	XML is quite more complex to learn than JSON.
It is simple to read and write.	It is more complex to read and write than JSON.
It is data-oriented.	It is document-oriented.
JSON is less secure in comparison to XML.	XML is highly secured.
It doesn't provide display capabilities.	It provides the display capability because it is a markup language.
It supports the array.	It doesn't support the array

Example :
 	
[
{
"name" : "Peter",
"employed id" : "E231",
"present" : true,
"numberofdayspresent" : 29
},
{
"name" : "Jhon",
"employed id" : "E331",
"present" : true,
"numberofdayspresent" : 27
}
]
	Example :
	
	<name>
	<name>Peter</name>
	</name>

JSON SYNTAX RULES

JSON syntax is derived from JavaScript object notation syntax:

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

JSON DATA - A Name and a Value

JSON data is written as name/value pairs, just like JavaScript object properties.

A name/value pair consists of a field name (in double quotes), followed by a colon, and followed by a value:

"firstName":"John"

JSON names require double quotes. JavaScript names do not.

In JSON, values must be one of the following data types:

- a string
- a number
- an object (JSON object)
- an array
- a boolean
- null

JSON OBJECTS

- A JSON object contains data in the form of key/value pair.
- The keys are strings and the values are the JSON types.
- Keys and values are separated by colon.
- Each entry (key/value pair) is separated by comma.
- JSON objects are written inside curly ‘{‘braces.
- Just like in JavaScript, objects can contain multiple name/value pairs:

```
{"firstName":"John", "lastName":"Doe"}
```

```
{
   "employee": {
      "name":      "sonoo",
      "salary":     56000,
      "married":    true
   }
}
```

In the above example, employee is an object in which "name", "salary" and "married" are the key. In this example, there are string, number and Boolean value for the keys.

JSON Object with Strings

The string value must be enclosed within double quote.

```
{
      "name":      "sonoo",
      "email":     "sonoojaiswal1987@gmail.com"
}
```

JSON Object with Numbers

JSON supports numbers in double precision floating-point format. The number can be digits (0-9), fractions (.33, .532 etc) and exponents (e, e+, e-,E, E+, E-).

```
{
"integer": 34,
"fraction": .2145,
"exponent": 6.61789e+0
}
```

JSON Object with Booleans

JSON also supports Boolean values true or false.

```
{
"first": true,
"second": false
}
```

JSON Nested Object Example

A JSON object can have another object also. Let's see a simple example of JSON object having another object.

```
{
    "firstName": "Sonoo",
    "lastName": "Jaiswal",
    "age": 27,
```

```
    "address": {
        "streetAddress": "Plot-6, Mohan Nagar",
        "city": "Ghaziabad",
        "state": "UP",
        "postalCode": "201007"
    }
}
```

JSON ARRAYS

- JSON array represents ordered list of values. JSON array can store multiple values. It can store string, number, Boolean or object in JSON array.
- In JSON array, values must be separated by comma.
- The **[**(square bracket) represents JSON array.

JSON Array of Strings

["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday"]

JSON Array of Numbers

[12, 34, 56, 43, 95]

JSON Array of Booleans

[true, true, false, false, true]

JSON Array of Objects

Just like in JavaScript, an array can contain objects:

```
{"employees":[
  {"firstName":"John", "lastName":"Doe"},
```

```
  {"firstName":"Anna", "lastName":"Smith"},
  {"firstName":"Peter", "lastName":"Jones"}
]}
```

CONVERTING A JSON TEXT TO A JAVASCRIPT OBJECT

JSON.parse ()

- A common use of JSON is to exchange data to/from a web server.
- When receiving data from a **web server**, the data is always a string.
- Parse the data with JSON.parse(), and the data becomes a JavaScript object.
- Use the JavaScript function JSON.parse() to convert text into a JavaScript object.

```
const obj = JSON.parse('{"name":"John", "age":30, "city":"New York"}');
```

JSON.stringify()

Convert a JavaScript object into a string with JSON.stringify().

```
const obj = {name: "John", age: 30, city: "New York"};
const myJSON = JSON.stringify(obj);
```

First, create a JavaScript string containing JSON syntax:

```
let text = '{ "employees" : [' +
'{ "firstName":"John" , "lastName":"Doe" },' +
'{ "firstName":"Anna" , "lastName":"Smith" },' +
'{ "firstName":"Peter" , "lastName":"Jones" } ]}';
```

Then, use the JavaScript built-in function JSON.parse() to convert the string into a JavaScript object:

```
const obj = JSON.parse(text);
```

Finally, use the new JavaScript object in your page:

```
<p id="demo"></p>

<script>

document.getElementById("demo").innerHTML =

obj.employees[1].firstName + " " + obj.employees[1].lastName;

</script>
```

JS ARROW FUNCTIONS

- Arrow functions were introduced in ES6.
- An **arrow function expression** is a compact alternative to a traditional function expression, but is **limited and can't be used in all situations.**
- Arrow functions provide you with an alternative way to write a shorter syntax compared to the function expression.
- Arrow functions allow us to write shorter function syntax.
- In javascript the '=>' is the symbol of an arrow function expression.
- Example,

```
// Traditional function expression
          const add = function(a, b) {
            return a + b;
          };

// Arrow function
          const addArrow = (a, b) => a + b;
```

In the example above, **add** and **addArrow** are equivalent functions that take two parameters and return their sum. The arrow function syntax is more concise and eliminates the need for the function keyword.

1. **Single Parameter:** If a function has only one parameter, you can omit the parentheses around the parameter list

```
// Traditional function expression
   const square = function(x) {
   return x * x;
  };

// Arrow function with a single parameter
  const squareArrow = x => x * x;
```

2. **No Parameters:** If a function has no parameters, you still need to use parentheses, but they will be empty

```
// Traditional function expression
const greet = function() {
  return "Hello, World!";
};

// Arrow function with no parameters
const greetArrow = () => "Hello, World!";
```

JS CALLBACK FUNCTIONS

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
- This technique allows a function to call another function.
- A callback function can run after another function has finished.

- A JavaScript callback is a function which is to be executed after another function has finished execution.

Why do we need Callback Functions?

JavaScript runs code sequentially in top-down order. However, there are some cases that code runs (or must run) after something else happens and also not sequentially. This is called asynchronous programming.

Callbacks make sure that a function is not going to run before a task is completed but will run right after the task has completed. It helps us develop asynchronous JavaScript code and keeps us safe from problems and errors.

In JavaScript, the way to create a callback function is to pass it as a parameter to another function, and then to call it back right after something has happened or some task completed.

JS PROMISES

Promises in JavaScript are an essential part of handling asynchronous operation

Creating a Promise:

A Promise is created using the Promise constructor, which takes a function as its argument.

function, often referred to as the **"executor,"**
takes **two parameters**: resolve and reject. These are functions that control the state of the Promise.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation or logic
  const operationSuccessful = true;

  if (operationSuccessful) {
    resolve("Operation completed successfully!");
```

```
  } else {
    reject("Operation failed!");
  }
});
```

Consuming a Promise:

Once a Promise is created, you can use the .then() and .catch() methods to handle the resolved (successful) or rejected (failed) states.

```
myPromise
  .then((result) => {
    console.log("Success:", result);
  })
  .catch((error) => {
    console.error("Error:", error);
  });
```

Promise.all():

The Promise.all() method allows you to wait for multiple promises to fulfill before executing a callback.

```
const promise1 = new Promise((resolve) => setTimeout(() => resolve("One"), 1000));
const promise2 = new Promise((resolve) => setTimeout(() => resolve("Two"), 2000));

Promise.all([promise1, promise2])

  .then((results) => {
    console.log("All promises fulfilled:", results);
  })

  .catch((error) => {
    console.error("Error:", error);
  });
```

JS Async-Await Functions

async and **await** are keywords in JavaScript that provide a more concise and readable way to work with asynchronous code.

Async Function:

An async function is a function that always returns a Promise.

The async keyword is placed before the function declaration.

```
async function fetchData() {
  return "Data fetched!";
}
```

Await Expression:

The await keyword can only be used inside an async function.

It pauses the execution of the function until the Promise is resolved, and then it resumes the execution and returns the resolved value.

```
async function fetchData() {
  const result = await someAsyncOperation();
  console.log(result);
}
```

JS Error Handling

Types of Errors

While coding, there can be three types of errors in the code:

Syntax Error: When a user makes a mistake in the pre-defined syntax of a programming language, a syntax error may appear.

Runtime Error: When an error occurs during the execution of the program, such an error is known as Runtime error. The codes which create runtime errors are known as Exceptions. Thus, exception handlers are used for handling runtime errors.

Logical Error: An error which occurs when there is any logical mistake in the program that may not produce the desired output, and may terminate abnormally. Such an error is known as Logical error

1. try...catch Statements:

The try...catch statement allows you to handle exceptions (errors) that occur in a block of code.

Use try to enclose the code that might throw an error and catch to handle the error.

```
try {
  // Code that might throw an error
  throw new Error("Something went wrong!");

}

catch (error) {
  // Handle the error
```

```
  console.error("Error:", error.message);
}
```

3. Throwing Custom Errors:

You can throw custom errors using the **throw** statement.

This is useful for signalling specific error conditions in your code.

```
function divide(a, b) {
  if (b === 0) {
    throw new Error("Division by zero is not allowed!");
  }
  return a / b;
}

try {
  const result = divide(10, 0);
  console.log("Result:", result);
}

catch (error) {
  console.error("Error:", error.message);
}
```