# <uses-permission android:name="android.permission.INTERNET" />

# Introduction to JSON

JSON (JavaScript Object Notation) is most widely used **data format for data interchange** on the web. This data interchange can happen between two computers applications at different geographical locations or running within same hardware machine.

The good thing is that JSON is a human and machine readable format. So while applications/libraries can parse the JSON data – humans can also look at data and derive meaning from it.

A JSON document may contains text, curly braces, square brackets, colons, commas, double quotes, and maybe a few other characters.

Primarily, JSON is built on two structures:

1. A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
2. An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

**JSON Example**

A sample JSON document looks like this:

|  |
| --- |
| //JSON Object    {      "employee": {          "id":           1,          "name":         "Admin",          "location":     "USA"      }  }    //JSON Array    {      "employees": [                   {                      "id":           1,                      "name":         "Admin",                      "location":     "USA"                   },                   {                      "id":           2,                      "name":         "User",                      "location":     "USA"                   },                   {                      "id":           3,                      "name":         "User2",                      "location":     "USA"                   }      ]  } |

As you can see, JSON data consists of name/value pairs. These name/value pairs reflect the structure of the data.

**Learn JSON**

In this tutorial, we will learn various concepts about JSON with examples. Such as:

* Difference with XML
* Syntax and data types
* How to read JSON data
* How to write JSON data
* Convert JSON to String and vice-versa etc.

# JSON Syntax

A JSON document may contain information separated by following separators/token.

1. ":" to separate name from value
2. "," to separate name-value pairs
3. "{" and "}" for objects
4. "[" and "]" for arrays

#### JSON name-value pairs example

Name-value pairs have a colon between them as in "name" : "value".

JSON names are on the left side of the colon. They need to be wrapped in double quotation marks, as in “name”, and can be any valid string. Within each object, keys need to be unique.

JSON values are found to the right of the colon. At the granular level, these need to be one of 6 simple data types:

1. strings
2. numbers
3. objects
4. arrays
5. booleans
6. null or empty

Each name-value pair is separated by a comma, so the JSON looks like this:

"name" : "value", "name" : "value", "name": "value" e.g.

|  |
| --- |
| {    "color" : "Purple",    "id" : "210"  } |

#### JSON object example

A JSON object is a key-value data format that is typically rendered in curly braces. A JSON object looks something like this:

|  |
| --- |
| {    "color" : "Purple",    "id" : "210",    "composition" : {      "R" : 70,      "G" : 39,      "B" : 89    }  } |

#### JSON array example

Data can also be nested within the JSON by using JavaScript arrays that are passed as a value using square brackets [ ] on either end of its array type.

JSON arrays are ordered collections and can contain values of differing data types.

|  |
| --- |
| {      "colors" :      [          {            "color" : "Purple",            "id" : "210"          },          {            "color" : "Blue",            "id" : "211"          },          {            "color" : "Black",            "id" : "212"          }      ]  } |

# JSON Data Types

At the granular level, JSON consist of 6 data types. First four data types (string, number, boolean and null) can be referred as **simple data types**. Other two data types (object and array) can be referred as **complex data types**.

1. string
2. number
3. boolean
4. null/empty
5. object
6. array

Let’s learn about them one by one.

| **DATA TYPE** | **DESCRIPTION** |
| --- | --- |
| String | Any sequence of characters, inserted between " and " (double quotes). e.g.   |  | | --- | | {    "color" : "Purple"  } | |
| Number | 1. Represented in base 10 with no superfluous leading zero (e.g. 000088). 2. These include digits between 0 and 9. 3. It can be a negative number (e.g. -10. 4. It can be a fraction (e.g. .5). 5. It can also have an exponent of 10, prefixed by e.  |  | | --- | | {    "number\_1" : 210,    "number\_2" : -210,    "number\_3" : 21.05,    "number\_4" : 1.0E+2  } | |
| Boolean | Value can be either true or false.   |  | | --- | | {    "visibility" : true  } | |
| Null | There is no value to assign. It can be treated as null.   |  | | --- | | {    "visibility" : true,    "popularity" : ,      //empty    "id" : 210  } | |
| Object | 1. An unordered set of name/value pairs inserted between {} (curly braces). 2. An object can contain zero or more name/value pairs. 3. Multiple name/value pairs are separated by a , (comma).  |  | | --- | | {    "visibility" : true,    "popularity" : "immense",    "id" : 210  } | |
| Array | 1. An ordered collection of values. 2. Begins with [ (left bracket) and ends with ] (right bracket). 3. It’s values are separated by , (comma).  |  | | --- | | {    "ids" : ["1","2","3"]  }    //or    {    "ids" : [          {"id" : 1},          {"id" : 2},          {"id" : 3}    ]  } | |

# JJSON Objects

1. JSON objects are very much like javascript objects.
2. JSON objects are written in key/value pairs.
3. JSON objects are surrounded by curly braces { }.
4. Keys must be strings, and values must be a valid JSON data type (string, number, object, array, boolean or null).
5. Keys and values are separated by a colon.
6. Each key/value pair is separated by a comma.

For example –

|  |
| --- |
| {      "name" : "Admin",      "age" : 36,      "rights" : [ "admin", "editor", "contributor" ]  } |

## Access object values

You can access object values in two ways :

#### 1) Using dot (.) notation

|  |
| --- |
| var author =  {                  "name" : "Admin",                  "age" : 36,                  "rights" : [ "admin", "editor", "contributor" ]              }    console.log( author.name );    //Output    Admin |

#### 2) Using bracket ([]) notation

|  |
| --- |
| var author =  {                  "name" : "Admin",                  "age" : 36,                  "rights" : [ "admin", "editor", "contributor" ]              }    console.log( author [ "name" ] );  console.log( author [ "age" ] );    //Output    Admin  36 |

## Looping object values

You can loop through object values using for loop, just like looping through an array.

|  |
| --- |
| var author =  {                  "name" : "Admin",                  "age" : 36,                  "rights" : [ "admin", "editor", "contributor" ]              }    for (x in author)  {      console.log(x + " - " + (author[x]);  }    //Output    name - Admin  age - 36  rights - admin,editor,contributor |

## Modify object values

To modify object values, use any of given two ways:

#### 1) Using dot (.) notation

|  |
| --- |
| var author =  {                  "name" : "Admin",                  "age" : 36,                  "rights" : [ "admin", "editor", "contributor" ]              }    author.name = "Lokesh";    console.log( author.name );    //Output    Lokesh |

#### 2) Using bracket ([]) notation

|  |
| --- |
| var author =  {                  "name" : "Admin",                  "age" : 36,                  "rights" : [ "admin", "editor", "contributor" ]              }    author["name"] = "Lokesh";  author["age"] = 35;    console.log( author [ "name" ] );  console.log( author [ "age" ] );    //Output    Lokesh  35 |

## Delete object values

Use the delete keyword to delete properties from a JSON object:

|  |
| --- |
| delete author.name; |

# JSON Schema

It’s often necessary for applications to validate JSON objects, to ensure that required properties are present and that additional constraints (such as a price never being less than one dollar) are met. Validation is typically performed in the context of JSON Schema.

JSON Schema is a grammar language for defining the structure, content, and (to some extent) semantics of JSON objects. It lets you specify metadata (data about data) about what an object’s properties mean and what values are valid for those properties. The result of applying the grammar language is a schema (a blueprint) describing the set of JSON objects that are valid according to the schema.

1. JSON Schema is expressed a schema, which itself is a JSON object.
2. JSON Schema is maintained at http://json-schema.org.
3. It describes your existing data format.
4. If offers clear, human-readable, and machine-readable documentation.
5. It provides complete structural validation, which is useful for automated testing and validating client-submitted data.

## JSON schema validation example

Let’s consider the schema is :

|  |
| --- |
| {     "$schema": "http://json-schema.org/draft-04/schema#",     "title": "Person",     "description": "A person",     "type": "object",     "properties":     {        "name":        {           "description": "A person's name",           "type": "string"        },        "age":        {           "description": "A person's age",           "type": "number",           "minimum": 18,           "maximum": 64        }     },     "required": ["name", "age"]  } |

and JSON object is :

|  |
| --- |
| {     "name": "John Doe",     "age": 35  } |

# JSON Array

1. The square brackets [ ] are used to declare JSON array.
2. JSON array are ordered list of values.
3. JSON array can store multiple value types.
4. JSON array can store string, number, boolean, object or other array inside JSON array.
5. In JSON array, values must be separated by comma.
6. Arrays in JSON are almost the same as arrays in JavaScript.

e.g.

|  |
| --- |
| {      "name" : "Admin",      "age" : 36,      "rights" : [ "admin", "editor", "contributor" ]  } |

#### Get value at specific index location in array

You can access the array values by using the index number:

|  |
| --- |
| x = myObj.rights[0];    //Output    admin |

#### Loop through array values

You can access array values by using a for-in loop:

|  |
| --- |
| for (i in myObj.rights) {      x = myObj.rights[i];      console.log(x);  }    //Output:    admin  editor  contributor |

#### Delete array item

Use the delete keyword to delete items from an array:

|  |
| --- |
| delete myObj.rights[1]; |

#### Update array value at index location

Use the index number to modify an array:

|  |
| --- |
| myObj.rights[1] = "blogger"; |

## Multi-dimentional array

We can store array inside JSON array, it is known as array of arrays or multidimensional array.

|  |
| --- |
| {      "name" : "Admin",      "age" : 36,      "rights" : [          {"roleName" : "admin", "roleIds" : [1,2,3] },          {"roleName" : "editor", "roleIds" : [4,5,6] },      ]   {"roleName" : "contributor", "roleIds" : [7,8,9]}  } |

#### Iterate over multi-dimentional array

|  |
| --- |
| for (i in myObj.rights) {        for (j in myObj.rights[i].roleIds) {            x = myObj.rights[i].roleIds[j];            console.log(x);      }  }    Output:    1  2  3  4  5  6  7  8  9 |

# JSON.parse()

As function name JSON.parse() suggests, this method converts a JSON string representation to a JavaScript value (JSON object).

It can optionally use a reviver function to perform a transformation on the resulting object before it is returned.

## Syntax

|  |
| --- |
| JSON.parse(text[, reviver]) |

Where –

1. **text** – The string to parse as JSON object.
2. **reviver** – A function that prescribes how the value originally produced by parsing is transformed, before being returned.

## JSON.parse() example

|  |
| --- |
| var jsonString = '{ "foo" : 1 }';  var obj = JSON.parse( jsonString );    var obj = JSON.parse('{ "x": 5, "y": 6 }');  //with space    //output    {foo: 1}    {x: 5, y: 6} |

## JSON.parse() reviver function example

This reviver function is called for each name-value pair in JSON string, and you can apply custom logic to find and modify any value before it is added to JSON object.

|  |
| --- |
| var obj = JSON.parse('{"1": 1, "2": 2, "3": {"4": 4, "5": {"6": 6}}}', (key, value) => {    if(value % 2 == 0)      return value \* 2; //If value is even number than multiply by 2    else      return value;    //else return original value  });    Output:    {1: 1, 2: 4, 3: {4: 8, 5: {6: 12}}} |

* **Log.e**: This is for when bad stuff happens. Use this tag in places like inside a catch statement. You *know* that an *error* has occurred and therefore you're logging an error.
* **Log.w**: Use this when you suspect something shady is going on. You may not be completely in full on error mode, but maybe you recovered from some unexpected behavior. Basically, use this to log stuff you didn't expect to happen but isn't necessarily an error. Kind of like a "hey, this happened, and it's *weird*, we should look into it."
* **Log.i**: Use this to post useful *information* to the log. For example: that you have successfully connected to a server. Basically use it to report successes.
* **Log.d**: Use this for *debugging* purposes. If you want to print out a bunch of messages so you can log the exact flow of your program, use this. If you want to keep a log of variable values, use this.
* **Log.v**: Use this when you want to go absolutely nuts with your logging. If for some reason you've decided to log every little thing in a particular part of your app, use the Log.v tag.