Day 5

Yogesh Yadav

JavaScript Variable Scope

JavaScript local variables : Local variables are the variables declared with in a function. These variables have local scope meaning these are available only inside the function that contains them.
Local variables are created when a function starts, and deleted as soon as the function completes execution.

```
function helloWorld()
{
    var greeting = "Hello";
    // The variable greeting is available in the function
    greeting = greeting + " JavaScript";
    alert(greeting);
}

helloWorld();

// The variable greeting is not available outside the function
// Error : 'greeting' is undefined

alert(greeting);
```

JavaScript global variables : Global variables are the variables declared outside a function. Global variables have global scope meaning all scripts and functions on the page can access them. The lifetime of a global variable starts with it's declaration and is deleted when the page is closed.

```
var greeting = "Hello";

function helloWorld()
{
    // The variable greeting is available in the function
    greeting = greeting + " JavaScript";
    alert(greeting);
}

helloWorld();
```

If you assign a value to a variable that has not been declared, it will automatically become a global variable, even if it is present inside a function.

If you assign a value to a variable that has not been declared, it will automatically become a global variable, even if it is present inside a function.

```
function helloWorld()
{
    // The variable greeting is not declared but a value is assigned.
    // So it will automatically become a global variable
    greeting = "Hello JavaScript";
}


helloWorld();


// Variable greeting is available outside the function


alert(greeting);
```

A local variable can have the same name as a global variable. Changing the value of one variable has no effect on the other. If the variable value is changed inside a function, and if a local version of the variable exists then the local variable gets modified. If the variable value is changed outside a function then the global variable gets modified.

```
var greeting = "This is from global Variable";

function helloWorld()
{
    var greeting = "This is from local variable";
    document.write(greeting + "<br/>");
}


// This line will modify the global greeting variable
greeting += "!!!";


helloWorld();
document.write(greeting);
```

Output :

```
This is from local variable
This is from global Variable!!!
```

Braces do not create scope in JavaScript : In the following example **otherNumber** is a global variable though it is defined inside braces. In many languages like C# and Java, braces create scope, but not JavaScript.

```
var number = 100;

if (number > 10)
{
    var otherNumber = number;
}

document.write(otherNumber);
```

Output : 100

Closures in JavaScript

What is a closure

A closure is an inner function that has access to the outer function's variables in addition to it's own variables and global variables. The inner function has access not only to the outer function's variables, but also to the outer function's parameters. You create a closure by adding a function inside another function.

JavaScript Closure Example

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    // This inner function has access to the outer function's variables & parameters
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    return add();
}

var result = addNumbers(10, 20);

document.write(result);
```

Output : Result is : 30

The following code Returns the inner function expression

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    // We removed the parentheses. This will return the
    // inner function expression without executing it.
    return add;
}


// addFunc will contain add() function (inner function) expression.
var addFunc = addNumbers(10, 20);
// call the addFunc() function and store the return value in result variable
var result = addFunc();
document.write(result);
```

Returning and executing the inner function

```
function addNumbers(firstNumber, secondNumber)
{
    var returnValue = "Result is : ";
    function add()
    {
        return returnValue + (firstNumber + secondNumber);
    }
    // We removed the parentheses. This will return the
    // inner function add() expression without executing it.
    return add;
}


// This returns add() function (inner function) definition
// and executes it. Notice the additonal parentheses.
var result = addNumbers(10, 20)();


document.write(result);
```

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

greet('Hi')('Tony');
```

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' ' + name);
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' '
+ name;
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

Global Execution Context

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' '
+ name;
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' '
+ name;
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

whattosay
'Hi'

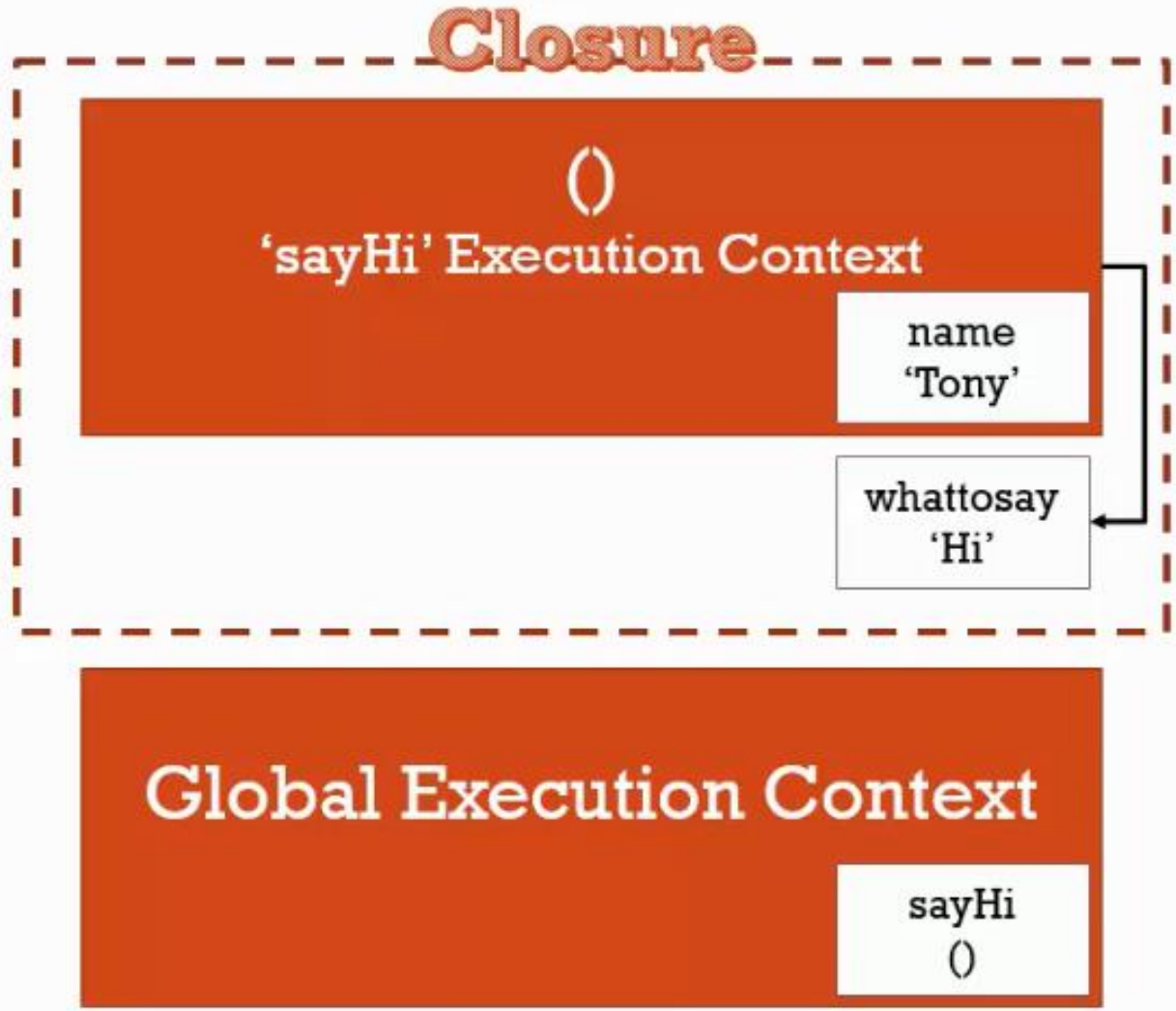
Global Execution Context

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' '
+ name;
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```

for whattosay variable go n check in scope chain out lexical environment

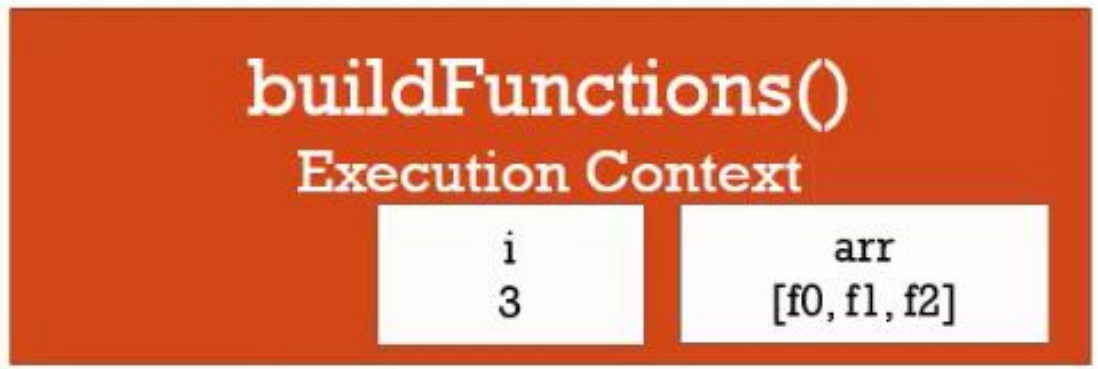
closing IN phenomena is called closure

```
function greet(whattosay) {

    return function(name) {
        console.log(whattosay + ' '
+ name;
    }

}

var sayHi = greet('Hi');
sayHi('Tony');
```



```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
             console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
        arr.push(function() {
            console.log(i);
        });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2];
```



```
function buildFunctions() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
       arr.push(function() {
             console.log(i);
       });
    }

    return arr;

}

var fs = buildFunctions();

fs[0]();
fs[1]();
fs[2]();
```

```
function buildFunctions2() {

    var arr = [];

    for (var i = 0; i < 3; i++) {
        arr.push(
            (function(j) {
                return function() {
                    console.log(j);
                }
            }(i))
        )

    }

    return arr;
}

var fs2 = buildFunctions2();

fs2[0]();
fs2[1]();
```

Using a global variable and incrementing it every time we click the button : The problem with this approach is that, since clickCount is a global variable any script on the page can accidentally change the variable value.

```
<script type="text/javascript">
    var clickCount = 0;
</script>
```

```
<input type="button" value="Click Me" onclick="alert(++clickCount);" />
```

Using a local variable with in a function and incrementing it by calling the function : The problem with this approach is that, click count is not incremented beyond 1, no matter how many times you click the button.

```
<script type="text/javascript">


    function incrementClickCount()
    {
        var clickCount = 0;
        return ++clickCount;
    }
</script>


<input type="button" value="Click
Me" onclick="alert(incrementClickCount());" />
```

Using a JavaScript closure : A closure is an inner function that has access to the outer function's variables in addition to it's own variables and global variables. In simple terms a closure is function inside a function. These functions, that is the inner and outer functions could be named functions or anonymous functions. In the example below we have an anonymous function inside another anonymous function. The variable incrementClickCount is assigned the return value of the self invoking anonymous function.

```
<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
            return ++clickCount;
        }
    })();
</script>
```

```
<input type="button" value="Click
Me" onclick="alert(incrementClickCount);" />
```

In the example above, in the alert function we are calling the variable incrementClickCount without parentheses. At this point, when you click the button, you get the inner anonymous function expression in the alert. The point I want to prove here is that, the outer self-invoking anonymous function run only once and sets clickCount variable to ZERO, and returns the inner function expression. Inner function has access to clickCount variable. Now every time we click the button, the inner function increments the clickCount variable. The important point to keep in mind is that no other script on the page has access to clickCount variable. The only way to change the clickCount variable is thru incrementClickCount function.

```
<script type="text/javascript">
    var incrementClickCount = (function ()
    {
        var clickCount = 0;
        return function ()
        {
            return ++clickCount;
        }
    })();
</script>
<input type="button" value="Click
Me" onclick="alert(incrementClickCount());" />
```

Arguments object

- The **JavaScript arguments object is a local variable available within all functions**. It contains all the function parameters that are passed to the function and can be indexed like an array. The length property of the arguments object returns the number of arguments passed to the function.

```
function printArguments()
{
    document.write("Number of arguments = " + arguments.length + "<br/>")
    for (var i = 0; i < arguments.length; i++)
    {
        document.write("Argument " + i + " = " + arguments[i] + "<br/>");
    }
    document.write("<br/>");
}

printArguments();
printArguments("A", "B");
printArguments(10, 20, 30);
```

```
Number of arguments = 0

Number of arguments = 2
Argument 0 = A
Argument 1 = B

Number of arguments = 3
Argument 0 = 10
Argument 1 = 20
Argument 2 = 30
```

Is it possible to pass variable number of arguments to a JavaScript function

Yes, you can pass as many arguments as you want to any javascript function. All the parameters will be stored in the arguments object.

```
function addNumbers()
{
    var sum = 0;
    document.write("Count of numbers = " + arguments.length + "<br/>")
    for (var i = 0; i < arguments.length; i++)
    {
        sum = sum + arguments[i];
    }
    document.write("Sum of numbers = " + sum);
    document.write("<br/><br/>");
}

addNumbers();
addNumbers(10, 20, 30);
```

Count of numbers = 0
Sum of numbers = 0

Count of numbers = 3
Sum of numbers = 60

Error handling in JavaScript

- **Use try/catch/finally to handle runtime errors in JavaScript.** These runtime errors are called exceptions. An exception can occur for a variety of reasons. For example, referencing a variable or a method that is not defined can cause an exception.

- **The JavaScript statements that can possibly cause exceptions should be wrapped inside a try block**. When a specific line in the try block causes an exception, the control is immediately transferred to the catch block skipping the rest of the code in the try block.

```
try
{
    // Referencing a function that does not exist cause an exception
    document.write(sayHello());
    // Since the above line causes an exception, the following line will not be
executed
    document.write("This line will not be executed");
}
// When an exception occurs, the control is transferred to the catch block
catch (e)
{
    document.write("Description = " + e.description + "<br/>");
    document.write("Message = " + e.message + "<br/>");
    document.write("Stack = " + e.stack + "<br/><br/>");
}
document.write("This line will be executed");
```

```
Description = 'sayHello' is undefined
Message = 'sayHello' is undefined
Stack = ReferenceError: 'sayHello' is undefined at Global code
(http://localhost:52212/HTMLPage1.htm:4:9)

This line will be executed
```

- **Please note :** A try block should be followed by a catch block or finally block or both.

  finally block is guaranteed to execute irrespective of whether there is an exception or not. It is generally used to clean and free resources that the script was holding onto during the program execution. For example, if your script in the try block has opened a file for processing, and if there is an exception, the finally block can be used to close the file.

```
try
{
    // Referencing a function that does not exist cause an exception
    document.write(sayHello());
    // Since the above line causes an exception, the following line will not be executed
    document.write("This line will not be executed");
}
// When an exception occurs, the control is transferred to the catch block
catch (e)
{
    document.write("Description = " + e.description + "<br/>");
    document.write("Message = " + e.message + "<br/>");
    document.write("Stack = " + e.stack + "<br/><br/>");
}
finally
{
    document.write("This line is guaranteed to execute");
}
```

Description = 'sayHello' is undefined
Message = 'sayHello' is undefined
Stack = ReferenceError: 'sayHello' is undefined at Global code (http://localhost:52212/HTMLPage1.htm:4:9)

This line is guaranteed to execute

Dates

- **To create date object in JavaScript use Date() constructor**

  The following example writes the current Date and Time to the web page
  document.write(new Date());

- If the **Date()** constructor is used without any arguments, it returns the current date and time. To create a date object with specific dates there are 2 ways.

  Creating a specific date object in JavaScript using a date string
  var dateOfBirth = new Date("January 13, 1980 11:20:00");
  document.write(dateOfBirth);

  You can also create a specific date object using number for year, month, day, hours, minutes, seconds, & milliseconds. The syntax is shown below.
  var dateOfBirth = new Date(year, month, day, hours, minutes, seconds, milliseconds);

  Example :
  var dateOfBirth = new Date(1980, 0, 13, 11, 20, 0, 0);
  document.write(dateOfBirth);

Timing events

- **In JavaScript a piece of code can be executed at specified time interval**. For example, you can call a specific JavaScript function every 1 second. This concept in JavaScript is called timing events.

- The global window object has the following 2 methods that allow us to execute a piece of JavaScript code at specified time intervals.

  setInterval(func, delay) - Executes a specified function, repeatedly at specified time interval. This method has 2 parameters. The **func** parameter specifies the name of the function to execute. The **delay** parameter specifies the time in milliseconds to wait before calling the specified function.

setTimeout(func, delay) - Executes a specified function, after waiting a specified number of milliseconds. This method has 2 parameters. The func parameter specifies the name of the function to execute. The delay parameter specifies the time in milliseconds to wait before calling the specified function. The actual wait time (delay) may be longer.

Let's understand timing events in JavaScript with an example. The following code displays current date and time in the div tag.

```
<div id="timeDiv" ></div>



<script type="text/javascript">
    function getCurrentDateTime()
    {
        document.getElementById("timeDiv").innerHTML = new Date();
    }


    getCurrentDateTime();
</script>
```

At the moment the time is static. To make the time on the page dynamic modify the script as shown below. Notice that the time is now updated every second. In this example, we are using setInterval() method and calling getCurrentDateTime() function every 1000 milli-seconds.

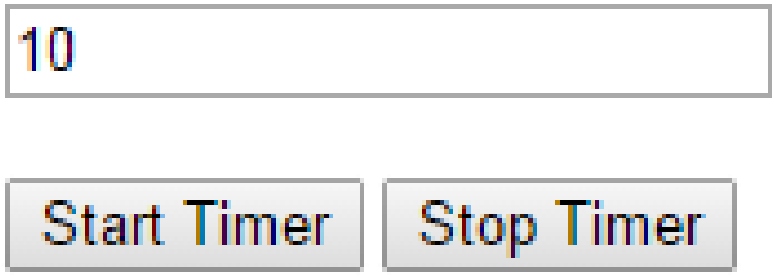
```
<div id="timeDiv" ></div>



<script type="text/javascript">
    setInterval(getCurrentDateTime, 1000);


    function getCurrentDateTime()
    {
        document.getElementById("timeDiv").innerHTML = new Date();
    }
</script>
```

clearInterval(intervalID) - Cancels the repeated execution of the method that was set up using **setInterval()** method. **intervalID** is the identifier of the repeated action you want to cancel. This ID is returned from **setInterval()** method. The following example demonstrates the use of **clearInterval()** method.

Starting and stopping the clock with button click : In this example, **setInterval()**method returns the **intervalId** which is then passed to **clearInterval()** method. When you click the **"Start Clock"** button the clock is updated with new time every second, and when you click **"Stop Clock"** button it stops the clock.

```
<div id="timeDiv" ></div>

<br />
<input type="button" value="Start Clock" onclick="startClock()" />
<input type="button" value="Stop Clock" onclick="stopClock()" />
<script type="text/javascript">
    var intervalId;


    function startClock()
    {
        intervalId = setInterval(getCurrentDateTime, 1000);
    }


    function stopClock()
    {
        clearInterval(intervalId);
    }


    function getCurrentDateTime()
    {
        document.getElementById("timeDiv").innerHTML = new Date();
    }


    getCurrentDateTime();
</script>
```

Countdown timer example : When we click **"Start Timer"** button, the value 10 displayed in the textbox must start counting down. When click **"Stop Timer"** the countdown should stop. When you click **"Start Timer"** again, it should start counting down from where it stopped and when it reaches ZERO, it should display **Done** in the textbox and function should return.



```
<input type="text" value="10" id="txtBox" />
<br /><br />
<input type="button" value="Start Timer" onclick="startTimer('txtBox')" />
<input type="button" value="Stop Timer" onclick="stopTimer()" />
<script type="text/javascript">
    var intervalId;


    function startTimer(controlId)
    {
        var control = document.getElementById(controlId);
        var seconds = control.value;
        seconds = seconds - 1;
        if (seconds == 0)
        {
            control.value = "Done";
            return;
        }
        else
        {
            control.value = seconds;
        }


        intervalId = setTimeout(function () { startTimer('txtBox'); }, 1000);
    }


    function stopTimer()
    {
        clearTimeout(intervalId);
    }
</script>
```

Object

- Objects in JavaScript are kind of two-faced.
- An object is an associative array (called *hash* in some languages). It stores key-value pairs.

- Name/Value Pair: A Name which maps to a unique value

- The name may be defined more than once, but only can have one value in any given **context**.

- That value may be more name/value pairs.

- Name=“abc”

- Object: A collection of name value pairs

```
menuSetup = {
     width:  300,
     height: 200,
     title: "Menu"
}
```

key	*value*
width	300
height	200
title	Menu

```
{
     CustomerID : 11 ,
     CustomerName : Guru99 ,
     OrderID : 111
}
```

Example of Key Value pairs

```
Address:
    {
        Street: ‘Main’,
        Number: 100
        Apartment:
        {
            Floor: 3,
            Number: 301
        }
    }
```

What is JSON

- JSON stands for JavaScript Object Notation. JSON is a lightweight data-interchange format. JSON is an easier-to-use alternative to XML.

- **Creating a JSON object** : Employee data can be stored in a JSON object as shown below.

- var employeeJSON = {
- "firstName": "Todd",
- "lastName": "Grover",
- "gender": "Male",
- "salary": 50000
- };
-

1. employeeJSON is a JSON object
2. In the curly braces we include the "name": "value" pairs, separated by commas
3. The name and value of a property are separated using a colon (:)
4. You can declare any number of properties

- If you want to represent the same data using XML, you may have XML that would look as shown below.
  <Employee>
- <firstName>Todd</firstName>
- <lastName>Grover</lastName>
- <gender>Male</gender>
- <salary>50000</salary>
- </Employee>

- Reading data from the JSON object : To read data from the JSON object, use the property names.
  var firstName = employeeJSON.firstName;

- **JSON Arrays :** What if you want to store more than one employee data in the JSON object. This is when JSON arrays can be used. A JSON array can contain multiple objects.

  To **create a JSON array**
  1. Wrap the objects in square brackets
  2. Each object must be separated with a comma

- **Creating a JSON array**
  var employeesJSON = [
- {
- "firstName": "Todd",
- "lastName": "Grover",
- "gender": "Male",
- "salary": 50000
- },
- {
- "firstName": "Sara",
- "lastName": "Baker",
- "gender": "Female",
- "salary": 40000
- }];

- **Reading from a JSON array :** To access the employee objects in the JSON array, use the object's index position.

  Retrieves the lastName of first employee object in the JSON array
  var result = employeesJSON[0].lastName;

  Retrieves the fistName of second employee object in the JSON array
  var result = employeesJSON[1].firstName;

- **Nested JSON object :** You can also store multiple employees in the JSON object by nesting them.

  Nested JSON object example :

  var employeesJSON = {

```
    "Todd": {
        "firstName": "Todd",
        "lastName": "Grover",
        "gender": "Male",
        "salary": 50000
    },
    "Sara": {
        "firstName": "Sara",
        "lastName": "Baker",
        "gender": "Female",
        "salary": 40000
    }
};
```

- Retrieves the gender of employee Todd
  var result = employeesJSON.Todd.gender;

  Retrieves the salary of employee Sara
  var result = employeesJSON.Sara.salary;

Converts JSON to a string.

- JSON.stringify() method converts a JSON object (or array) into a JSON string.

- **converts a JSON string to a JSON array**. JSON.parse() method converts a JSON string to JSON array.