JAVASCRIPT – III

Topics:

- Switch statements
- Functions
- Arrays (Lists in Python)

Switch Statement

We can use multiple if…else statements, as in the previous session, to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

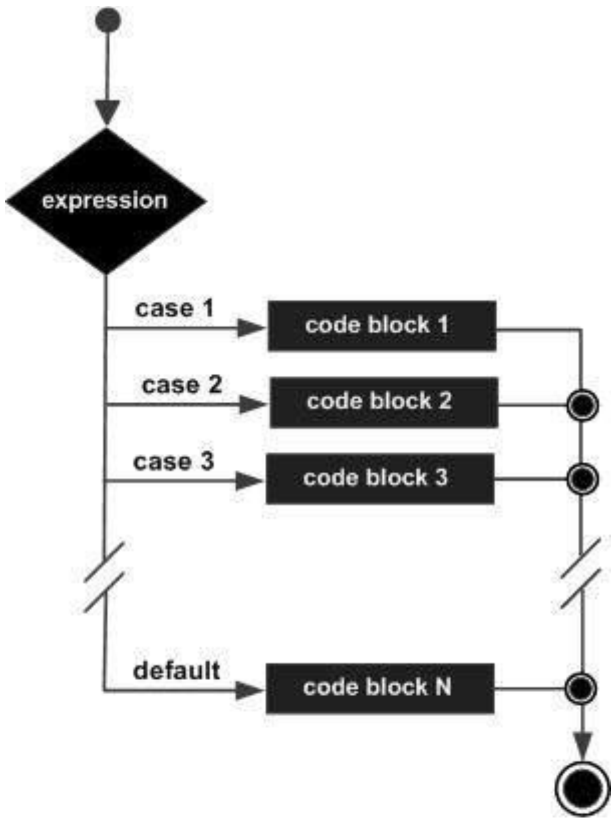
We can use a **switch** statement which handles exactly this situation, and it does so more efficiently than repeated **if…else if** statements.

Syntax:

The objective of a **`switch`** statement is to give an expression to evaluate several different statements to execute based on the value of the expression. The interpreter checks each **`case`** against the value of the expression until a match is found. If nothing matches, a **`default`** condition will be used.

```
//switch statements

switch (expression) {
    case x:
        // logic
        break
    case y:
        // different logic
}
```



`expression`: An expression whose result is matched against each **`case`** clause.

The break statement indicates the end of a particular case. If they were omitted, the interpreter would continue executing each statement in each of the following cases.

```
var userOption = 'A' // A, B, C, D

switch (userOption) {

    case 'A':
        console.log('User Chose Option A')
        break

    case 'B':
        console.log('User Chose Option B')
        break

    case 'C':
        console.log('User Chose Option C')
        break

    default:
        console.log('User Chose an Invalid Option')
}
```

The variable 'A' will be matched with every case inside the switch statement and once the program gets to case B, then it will print the statement in the console.log

If we do not put **`break`** statement at B then once the program reaches case B, as it has found the match it will print starting from case B to the default case. This is the default behavior of the **`switch`** statement.

If we want to call one function then we need to use break statements, else if we want all the options to be used from the matched option, then if we do not use break statement everything after matched option will be printed.

If we put default in between cases, then JavaScript will drop us back to the default if it can't find a match.

Functions:

Function Declaration:

A function is a group of reusable code which can be called anywhere in your program. This eliminates the need of writing the same code again and again. It helps programmer in writing modular codes. Functions allow a programmer to divide a big program into a number of small and manageable functions.

Function Definition:

Before we use a function, we need to define it. the most common way to define a function in JavaScript is by using the function keyword, followed by a unique function name, a list of parameters (that might be empty), and a statement block surrounded by curly braces.

Syntax:

```
function myFunction (param1, param2) {
    // your logic
    return param1 * param2
}

myFunction()
```

EX:

```
function myFunction (a, b = 30) {
    console.log(a * b)
}

myFunction(10, 15)
```

OUTPUT:

150

The output is 150 because in the first argument the value of b = 15.

Function Expression:

Instead of providing a name to the function you provide a variable to it.

```
var add = function(num1, num2) {
    return num1 + num2
}
```

The change here is, add is a variable, but in the previous case, we were only giving name to our function. But in this case, we are creating a variable and saving the function definition inside the variable. So, to call the function,

```
console.log(add(20, 30))
```

OUTPUT:

50

Scope:

Scope is the accessibility of variables, functions and objects in some particular part of your code during runtime. In other words, scope determines the visibility of variables and other resources in areas of your code.

Scope provides some level of security to your code. One common principle of computer security is that users should only have access to the stuff they need at a time.

In JavaScript language there are two types of scopes:

- Global Scope
- Local Scope

Variable defined **inside a function** are in local scope while variables defined outside of a function are in the global scope. Each function when invoked creates a new scope.

When you start writing a JavaScript in a document, you are already in the Global Scope. There is only one global scope throughout a JavaScript document. A variable is in the global scope if it’s defined outside of a function.

Ex: 01

INPUT:

```
var name = 'Yash'
// this logs global variable
function displayName() {
    console.log(name)
}
displayName()
```

OUTPUT:

```
Yash
```

Variables defined inside a function are in the local scope. And they have a different scope for every call of that function. This means that variables having the same name can be used in different functions. This is because those variables are bound to their respective functions, each having different scopes, not accessible in other functions.

Ex: 02

INPUT:

```
var name = 'Yash'
function displayName(){
    var name = 'Test'
// this logs local variable
    console.log(name)
}
displayName()
```

OUTPUT:

```
Test
```

If we want to use the global variable then we will use the window.object to get the global variable.

Ex: 03

INPUT:

```
var name = 'Yash'
function displayName(){
    var name = 'Test'
// this logs global variable
    console.log(window.name)
}
displayName()
```

OUTPUT:

```
Yash
```

In javascript there are 3 global objects, out of those 2 are used frequently.

- Window Object
  - This represents the entire browser window.
- Document
  - All the changes related to html will be visible in the document.

When we define a global variable, it gets attached to the window object. In the console.log when we inspect 'window' and expand upon it, then we see the name property. It shows 'Yash' which we have declared.

The displayName is also attached to the window object, which can be called using **`displayName()`** or **`window.displayName()`**

Array OR Lists:

Objects allow you to store keyed collections of values.

But quite often we find that we need an ordered collection, where we have a 1st, a 2nd, a 3rd element and so on. For example, we need that to store a list of something: users, goods, HTML elements etc.

It is not convenient to use an object here, because it provides no methods to manage the order of elements. We can't insert a new property "between" the existing ones. Objects are just not meant for such use.

There exists a special data structure named Array, to store ordered collections.

Array or lists are collection of data.

```
var list = [1,2,3]
var mixedList = [1,2,3,"Name",'text']
```

Arrays are used to group similar data types.

Functions of List:

`list.length()`	this gives the total count of elements in the list.
`list.reverse()`	it reverses the elements of given array
`list.push()`	pushes the element 10
`list.pop()`	pops the last element

`list.unshift()`	adds one or more elements in the beginning of the given array
`list.shift()`	it removes and returns the first element of an array

https://www.javatpoint.com/javascript-array

https://javascript.info/array

Callback Function:

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.

```
function specificFunction() {
    console.log('This is coming from function named specificFunction')
}

function anyFunction(params) {
    console.log('This is coming from function anyFunction')
    console.log('params is ', params)
    // params()
}

anyFunction(specificFunction)
```

Explantion:

we have defined a function *anyFunction* and we put *params* as a parameter. In this function we will put a console.log as ‘This is coming from anyFunction’.

This function can be called using anyFunction()

```
function anyFunction(params) {
    console.log('this is coming from function named anyFunction')
}
anyFunction()
```

This will give us an output as

This is coming from function named anyFunction.

Just as in Python, we have declared a function and called it.

We use ***def function()*** in Python and in JavaScript we will be using ***function functionName().***

Now, lets define another function ***specificFunction()*** and lets just console.log inside this function as 'this is coming from function named specificFunction'.

We can pass this (specificFunction) function as parameter or an argument to ***anyFunction()***.

```
function specificFunction() {
    console.log('This is coming from function named specificFunction')
}

function anyFunction(params) {
    console.log('This is coming from function anyFunction')
}

anyFunction(specificFunction)
```

Now, the parameter params will actually refer to specificFunction when we try to access params.

If we add console.log in anyFunction, we would get the value of params.

```
function specificFunction() {
    console.log('This is coming from function named specificFunction')
}

function anyFunction(params) {
    console.log('This is coming from function anyFunction')
    console.log('params is ', params)
}

anyFunction(specificFunction)
```

OUTPUT:

```
This is coming from function anyFunction

Params is f specificFunction(){
        console.log('This is coming from function named specificFunction')
}
```

The function is getting passed. So, we if put params() in anyFunction(), then we would get an OUTPUT as:

This is coming from function named specificFunction.

The function can be anything, that can be called using our params().

When we say ***var abc = 10**;* in this ‘abc’ is an identifier which is used to identify a variable.

Just we pass in parameters to a function, we have used a function as parameter. This is called ***callback function***.

Anonymous Function

An anonymous function is a function without a name. An anonymous function is often not accessible after its initial creation.

```
operation (function(){console.log("Anonymous function")})
```

In this example, anonymous function has no name between the function keyword and parentheses () because we need to call the anonymous function later, we assign the function to the **show** variable.

In the above example we are using the anonymous function as arguments of other functions. We pass an anonymous function into the **operation()** function.