Selenium with JavaScript, TypeScript

- QtpSudhakar

Introduction

About Trainer

- Sudhakar Kakunuri known as Qtp Sudhakar
- Worked as a Tester, Senior, Lead, Manager, Architect and a Trainer for Test Automation
- Working as a Freelance Test Automation Architect
- Blogs at QtpSudhakar.com on various Test Automation concepts
- Started career on WinRunner -> QTP/UFT -> Selenium -> Appium, Protractor, UFTPro, Katalon Studio…etc.
- Authored “Cracking the QTP Interview” book published by TATAMcGraw Hill
- Gives training and support on Various Test Automation Tools

What will be covered in this Course….

- VSCode IDE
- JavaScript
- TypeScript
- Starting with Web Driver
- Locators
- Working With Elements
- Mouse and Keyboard Operations
- Handling Wait Time

What is NodeJS?

- A node is runtime environment for executing java script code
- Every browser has Java Script Engine that converts JavaScript code into Machine Code
- Edge has chakra, Firefox has Spider Monkey, Chrome has v8 engine to execute JavaScript
- In 2009 Ryan Dahl the creator of NodeJS embedded the V8 JavaScript engine in C++ program and called that program NODE
- Similar to browsers, NODE is also a runtime environment for JavaScript code
- It also has several objects that provide environment for JavaScript
- These Objects are different from Browser Environment
- Node gives us some additional functionality that is not available in browsers
- Node is not a programming language or framework

What is TypeScript?

- JavaScript is an interpreted language which executes in browser
- It's a scripting language is used for created client side validations
- TypeScript is an open source programming language developed and maintained by Microsoft
- It is a super set of JavaScript and adds strict syntax and typing to the JavaScript
- JavaScript is not strongly typed language
- It checks for the type while writing the code not while running the code
- TypeScript doesn't run in browser
- It is compiled to JavaScript and finally executes in browser

JavaScript Topics

- Variables
- Datatypes
- Constants
- Arrays
- Objects
- Prototypes
- Conditions
- Loops
- Functions
- Objects
- Exception Handling
- Callback
- Promisses
- Async Await
- Modules
- File Handling

JavaScript Introduction

- JavaScript designed by Brendan Eich and developed by NetScape, Mozilla and ECMA
- It is mainly created for client side scripting language
- Later server side JavaScript is introduced with NodeJS
- The first version released in 1995 and latest version is 10
- It follows standards provided by ECMAScript
- ActionScript and Jscript are the other languages follows ECMA standards
- But JAVA SCRIPT remains best-known implementation of ECMAScript

Datatypes and Variables

- By default we don't need to declare a variable in javascript
- We can declare variable using **var** or **let**
- var declares global variable and let declares local scope variable
- We can assign any type of data (Boolean, null, undefined, Number, String and Object)
- A JavaScript identifier must start with a letter, underscore (_), or dollar sign ($)
- A variable declared using the var or let statement with no assigned value specified has the value of undefined.
- We can declare constants using **const**
- These are read only variables
- We can use "use strict" to make javascript to expect declarations in scope where it is used

Operators

- All are same like java
- == compares data
- === compares data and type
- ? Ternary operator also works same like java

```
var x = 10;
console.log(x==10);
console.log(x == "10");
console.log(x === "10"); //compares value and type


var browser;
var browsername = browser == undefined ? 'chrome' : browser;
console.log(browsername);
```

Arrays

- Unlike Java, JavaScript Arrays are dynamic
- Every variable is an object
- Array object will have some methods
- You can specify values in square brackets or use new Array(values) to create array

```
var userDetails = ["sudhakar","k", "33"];

console.log(userDetails[0]);
console.log(userDetails[1]);
console.log(userDetails[2]);

console.log(userDetails.length);

var userData = new Array("sudhakar","k", "33");
console.log(userData.length);
console.log(userData);
```

Continued…

- You can use push method to add some more data later
- concat returns a new array but not modify existing array
- push and concat accept multiple values

```
userData.push("trainer","asd");
console.log(userData);
var nar = userData.concat('four');
console.log(userData);
console.log(nar);
```

- we can add elements to beginning of array using unshift

```
userData.unshift('zero');
console.log(userData);
```

Continued… Multi Dimensional Arrays

```
var ar = [
  ['apple', 'orange', 'pear'],
  ['carrots', 'beans', 'peas'],
  ['cookies', 'cake', 'muffins', 'pie']
];

console.log(ar[0][1]);
console.log(ar[1][0]);
console.log(ar[2][3]);

console.log(ar.length);
console.log(ar[0].length);
console.log(ar[2].length);
```

Get Datatypes of Variables

- TypeOf returns string, number, boolean, undefined, function, or object
- When there is no value assigned it then it returns undefined
- For arrays it returns object
- String(), Number(), Boolean(), and Object() functions can be used to convert data from one type to another
- parseInt and parseFloat functions return numbers from strings that start with numeric data
- When JavaScript can't convert the passed string to a number, NaN is returned

If Conditions

```
/**
 * find number is odd number or even
 */

var n = 10;

if (n / 2 == 1) {
console.log(n + " is odd number")
} else {
console.log(n + " is even number")
}
```

```
/**
 * find which number bigger
 */

var x = 30;
var y = 30;

if (x>y) {
console.log("x is bigger")
} else if(x<y){
console.log("y is bigger")
}else{
console.log("Both are equal")
}
```

Switch Case

```
/**
 * Find index by month
 */

var mName = "mar";

switch (mName) {

case "mar":
  console.log(3);
  break;
case "feb":
  console.log(2);
  break;
case "apr":
  console.log(4);
  break;
case "jan":
  console.log(1);
  break;
default:
  console.log("Month not found");
  break;
}
```

Loops

```
/**
 * print values of an array
 */

var x = [ 10, 20, 30 ];

for (var i = 0; i < x.length; i++) {
console.log(x[i]);
}

for(var val of x){
console.log(val);
}

for(k in x){
console.log(x[k]);
}
```

```
/*
 * print values from 1 to 9
 */

var x = 1;

while (x < 10) {
console.log(x);
x++;
}

var x = 0;
do {
console.log(x);
x++;
} while (x < 10);
```

Functions

- Function is a block of reusable code
- it can be created in two ways
  - using function fName(parameters) { }
  - using var variable = function (parameters) { } //function expression
- We can call function with or without passing values
- it doesn't throw error instead NAN will return in case if we dont pass values
- We can even pass values more than parameter count. Java Script Ignores those values
- By default JavaScript moves functions to top scope
- That's why we can call functions before or after defining function
- It is called function hoisting
- When a function is created using function expression then it is not hoisted

Function Code Demo

```
function add(a,b){
return a+b;
}

console.log(add(2,3));

function restParam(...x){
 console.log(x.length); // Logs the number of arguments passed
}

restParam(1,2,3,4);

// parameters are references are values.

function updateArr(x){
x[2]="updated";
}

var a = [10,20,30];
updateArr(a); //value of a will be updated
console.log(a);
```

Arrow Functions

```
// Normal Function
var anon = function (a, b) { return a + b };
console.log(anon(2,2));

// we could write the above example as:
var anon1 = (a, b) => a + b;
console.log(anon1(2, 3));

// or

var anon2 = (a, b) => { return a + b };
console.log(anon1(2, 4));

// if we only have one parameter we can loose the parentheses
var anon3 = a => a;
console.log(anon3(2));

// and without parameters
var anon4=() => { console.log("nothing")};
anon4();
```

Objects

- All JavaScript values, except primitives, are objects.
- JavaScript defines 5 types of primitive data types:
- when a variable is assigned with string, number, boolean, null, undefined values then that becomes premitive
- Objects are variables too. But objects can contain many values.
- The values are written as name : value pairs (name and value separated by a colon).
- arrays are like lists and objects are like maps

```
var person = {
firstName : "sudhakar",
lastName : "k",
age : 33
};

console.log(person.firstName);
console.log(person.age);
console.log(person['lastName']);
```

```
for ( var d in person) {
console.log(d);
console.log(person[d]);
}

for (var i = 0; i < person.length;
i++) {
  console.log(person[i]);
}
```

Objects with Functions

- We can add or delete properties to objects after they are created
- We can also add functions to objects

```
person.nationality = "Indian"; // adding new property
console.log(person.nationality);

delete person.age;
console.log(person.age); //returns undefined
```

```
var person = {
firstName : "sudhakar",
lastName : "k",
age : 33,

act:function(){
console.log(this.firstName+" is acting");
}
};

person.act();
```

Object Constructors

- Function constructs objects is a constructor
- You cannot add a new property to an object constructor
- you can add a new property to an existing object
- this keyword refers to the object it belongs to
- In a method, this refers to the owner object
- Alone, this refers to the global object
- In a function, this refers to the global object.

Object Constructors

```
function Person(first, last, age, eye) {
  this.firstName = first;
  this.lastName = last;
  this.age = age;
  this.eyeColor = eye;
}

var myFather = new Person("subba", "reddy", 50,"red");

Person.nationality = "English";
console.log(Person.nationality); //return english

var p1 = new Person("sudha", "reddy", 50,"red","indian");

console.log(p1.nationality); //return undefined
```

Reuse Objects of one file in other

- We can reuse objects of one file from other file
- We can use import and export for that purpose

```
//DemoMaths.js
var DemoMaths = function() {

  this.add = function(x, y) {
return x + y;
  }

  this.mul = function(x, y) {
return x * y;
  }
}

module.exports = new DemoMaths();
```

```
//ReuseJSFileMethods.js
var dm = require('./DemoMaths');

console.log(dm.add(2,3));
console.log(dm.mul(2,3));
```

Prototypes

- All JavaScript objects inherit properties and methods from a prototype.
- Date objects inherit from Date.prototype. Array objects inherit from Array.prototype.
- Person objects inherit from Person.prototype.
- The Object.prototype is on the top of the prototype inheritance chain:
- Date objects, Array objects, and Person objects inherit from Object.prototype.
- The JavaScript prototype property allows you to add new properties and methods to object constructors
- Only modify your own prototypes
- It doesn’t add new property while creating objects
- It adds a property to all objects

Prototypes Code

```
function Person(first, last, age, eyecolor) {
this.firstName = first;
this.lastName = last;
this.age = age;
this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";

Person.prototype.name = function() {
return this.firstName + " " + this.lastName;
};

Console.log(Person.prototype.nationality);
```

Asynchronous

- JavaScript is always synchronous and single-threaded.
- JavaScript is Asynchronous some situations like AJAX calls
- When it is Asynchronous it takes requests one by one and executes whatever is executable right away
- Synchronous: When a function sends a database request it doesn't execute other code until it gets a response.
- Asynchronous: When a function sends a database request it will wait for response and allows other code to execute queue and event loop
- Because we call many functions as part of automation they doesn't execute in specific order
- But for Test Automation the statements must be executed step by step

Callback

- In Javascript Functions are first class objects
- We can pass functions as arguments to another function
- This will be helpful to separate logic outside and also to execute functions one after another
- This concept is called callback

```
function calc(a, b, act) {
    switch (act) {
        case "add":
            return a + b;
        case "multiply":
            return a * b;
    }
}
console.log(calc(10, 20, "add"));
```

```
add = (a, b) => a + b;
multiply = (a, b) => a * b;

function calculate(a, b, callback)
 {
        return callback(a, b);
}
console.log(calculate(2, 3, add));
```

Promises

- Using call back we can execute one after other
- But promise gives a feature to execute based on status of other
- These are create to handle asynchronous operations of javascript
- A promise will have three states
  - Pending
  - Resolved
  - Rejected
- It executes the other statement when it is resolved or rejected
- We can use .then() to wait for current request to be completed by holding all other.
- We can pass a callback to then() to execute after promise resolve or reject
- We can also use async and await to make synchronous program execution
- But we can use async and await only with function combination
- Functions must started with async keyword and use await before for every promise call statement in functions

Promise Example

```
const users = [
    { name: "user1", age: 25 },
    { name: "user2", age: 30 } ];

function getUsers() {
    setTimeout(() => {
        users.forEach((user) => {
            console.log(user.name + ":" + user.age);
        }
        );}, 1000);}
function createUsers(user) {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            users.push(user);
            const userCreated = true;
            if (userCreated) {
                resolve();
            } else {
                reject('user not created');
            }}, 2000);
    });}
createUsers({ name: "user3", age: 25 }).then(getUsers).catch(err => console.
log(err));
```

Async Await Example

```
//this will be asynchronous if you use async await
//all the tasks in micro-
task queue will be executed before the tasks in message queue if
written outside function with sync await
async function ex1() {
    // Say "Hello."
    await console.log("Hello.");
    // Say "Goodbye" two seconds from now.
    await setTimeout(async function () {
        await console.log("waiting is over");
    }, 5000);

    // Say "Hello again!"
    await console.log("Hello again!");
}
ex1();
```

Closure

- A closure is an inner function that has access to the outer (enclosing) function's variables—scope chain.
- The closure has three scope chains: it has access to its own scope, outer function's variables and global variables
- A closure is a function having access to the parent scope, even after the parent function has closed.

```
var countForMe = function () {
    let counter = 0;
    counter += 1;
    return counter;
}

console.log(countForMe()); //1
console.log(countForMe()); //1
```

```
var add = (function () {
    let counter = 0;
    console.log("fromclosure:" + c
ounter);
    return function () { counter +
= 1; return counter }
})();

console.log(add()); //1
console.log(add()); //2
```

Modules

- A module is a reusable piece of code that encapsulates implementation details and exposes a public API so it can be easily loaded and used by other code.
- Modules let the developer define private and public members separately
- modules are like Classes in OOPS
- These are used for connecting two JavaScript programs together to call the functions written in one program
- You need to import a library to use any of it
  - const lib = require('./library')
- You need to export to import any library module
  - module.exports = { module1,module2 }
  - module.exports = new DemoMaths();

Modules Sample

```
//DemoMaths.js
var DemoMaths1 = function () {

    this.add = function (x, y) {
        return x + y;
    }

    this.mul = function (x, y) {
        return x * y;
    }
}

module.exports = new DemoMaths1();
```

```
//ReuseJSFileMethods.js
var dm = require('./DemoMaths');

console.log(dm.add(2,3));
console.log(dm.mul(2,3));
```

Exception Handling

- Exception handling is similar to java with try catch finally blocks

```
try {
  var x = "10";
  var y = "x";
  console.log(eval(x));
  console.log(eval(y));
  console.log(eval("z="));
} catch (error) {
  console.error(error);
  // SyntaxError: Unexpected end of input
}
```

```
try {
  eval('5 + / 3'); // will raise SyntaxError exception
} catch (e) {
  // Compare as objects
  if (e.constructor == SyntaxError) {
// Get the error type as a string for reporting and storage
console.log(e.constructor.name); // SyntaxError
  }
}
```

Exception Types

- There are 6 types of JS errors
  - Eval Error : Deprecated in newer versions. Error from Eval function
  - Range Error : value is not in the range of allowed values
  - Reference Error : Variable reference not exist
  - Syntax Error : Syntax not followed in code
  - Type Error : Value is not expected type to apply method
  - URI Error : Error while decoding URI (URL)

```
n = 1.99999
console.log(n.toFixed(200)); //range error
z = x + y; //reference error
eval("x=") //syntax error
n.toUpperCase() //type error
decodeURI("%");   // URI error
```