WSA | Forward looking IT finishing school

Debugging Techniques

JavaScript

JavaScript Errors

(Capturing errors using JavaScript constructs)

Errors

In JavaScript three types of error are:

- **Syntax Error:** Occurs at compile / interpreting time
- **Run Time Errors:** Happens during execution. Exceptions and handling happens here
- **Logical Error:** Occurs when we make mistake in logic

The try-catch

Syntax:

```
try
{
   //statements
}
catch (error)
{
   //statements
}
```

The Statement try is used to enclose and test parts of the program where some problem is expected. If an exception is encountered the control is shifted to catch block.

To the catch block the problem is returned in form of error object, which has two properties:
`Name`   : Name of the error (category)
`Description` : Details about the error

The try-catch example

```
<script>
     function errorFunc()
     {
        try {
               // Write some junk
                somejunk();
            }

        catch(e){
               alert("Error Name    :  "   + e.name);
               alert("Error Message :  "   + e.message);
            }
      }
 </script>
 <body>
     <button onclick=“errorFunc()”>Test Error Function</button>
</body>
```

The Error Object

Methods	Description
`RangeError`	A number "out of range" has occurred
`ReferenceError`	An illegal reference has occurred
`SyntaxError`	A syntax error has occurred
`TypeError`	A type error has occurred
`URIError`	An error in encodeURI() has occurred

The finally statement

```
Syntax:

try
{
   // statements
}
catch (error)
{
   // statements
}
finally
{
   // statements
}
```

The finally clause is used to execute statements after the end of try block, whether or not an exception occurred within the try block.

The throw statement

```
Syntax:

try
{
   // statements
   throw “statements”;
}

catch (error)
{
   // statements
}
```

The throw statement allows to create user defined conditions for exceptions.

JavaScript Throw Example

```
<script>
    function errFunc()
       {
          var x = Number(prompt("enter x value"));
          var y = Number(prompt("enter y value"));

          try{
             if ( y == 0 ){
                throw( "Divide by zero error." );
             }
           else
             {
                var z = x / y;
                document.write("z ="+z+"<br>");
             }
          }
          catch ( e ) {
             alert("Error:  " + e );
          }
       }
  </script>
```

Exercise

- Write a JavaScript program to enter the age of any person and if age is less than 18 then throw an exception “not eligible for voting”
- Write a JavaScript program to enter the number between 5 to 20. If the number is not within range then throw an user defined exception

Debugging

(Debugging the running program using debugger provided by browser)

Debugging

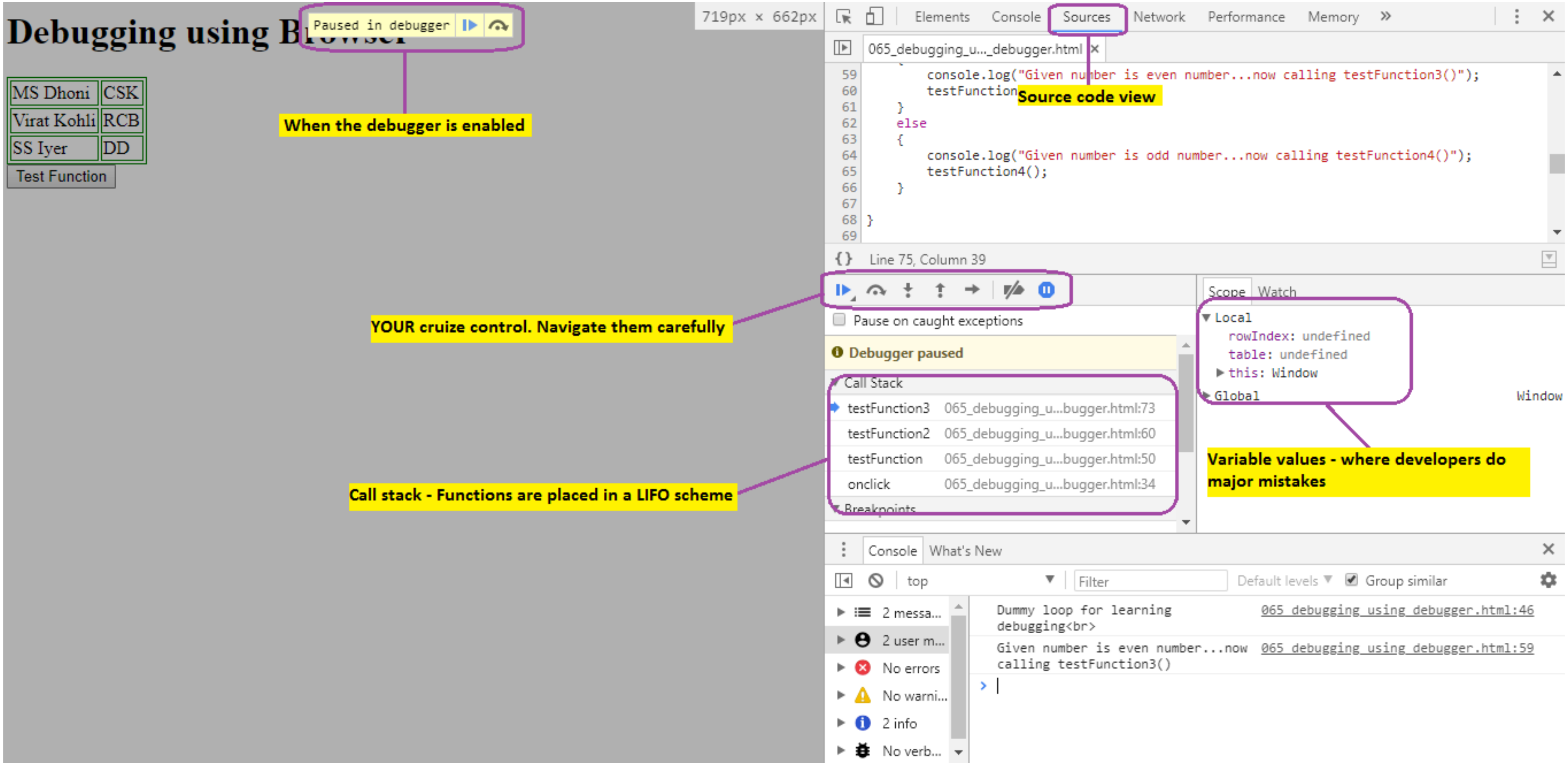
- All modern browsers have a built-in debugger. In our case Chrome browser is provided as a case study
- These debuggers provide facility to walk through the program during run-time
- It will give you the live snapshot of the program, even alter the flow of the program by forcing variable values
- By carefully investigating all the facilities provided by the debugger developers can empower themselves
- However please note before getting into this run-time debugging, ensure previous steps (Requirements understanding, Algorithm Design, Pseudo-code, Dry-run) are followed well.
- Best way to solve a problem is to avoid them ☺

The “debugger” keyword

- The debugger keyword stops the execution of JavaScript and calls the debugging function
- To view the debugger window press F12

```
<script>
  var x = 4 * 5;
  debugger; //stop executing before its executes next line
  document.getElementById("ex").innerHTML = x;
 </script>
```

The Debugger window



MS Dhoni	CSK
Virat Kohli	RCB
SS Iyer	DD

```
        console.log("Given number is even number...now calling testFunction3()");
        testFunction
    }
    else
    {
        console.log("Given number is odd number...now calling testFunction4()");
        testFunction4();
    }

}
```

Line 75, Column 39

Debugger paused

Call Stack

testFunction3	065_debugging_u...bugger.html:73
testFunction2	065_debugging_u...bugger.html:60
testFunction	065_debugging_u...bugger.html:50
onclick	065_debugging_u...bugger.html:34

Breakpoints

```
▼ Local
    rowIndex: undefined
    table: undefined
  ▶ this: Window
▶ Global                                   Window
```

```
Dummy loop for learning           065_debugging_using_debugger.html:46
debugging<br>
Given number is even number...now 065_debugging_using_debugger.html:59
calling testFunction3()
>
```

Source code view

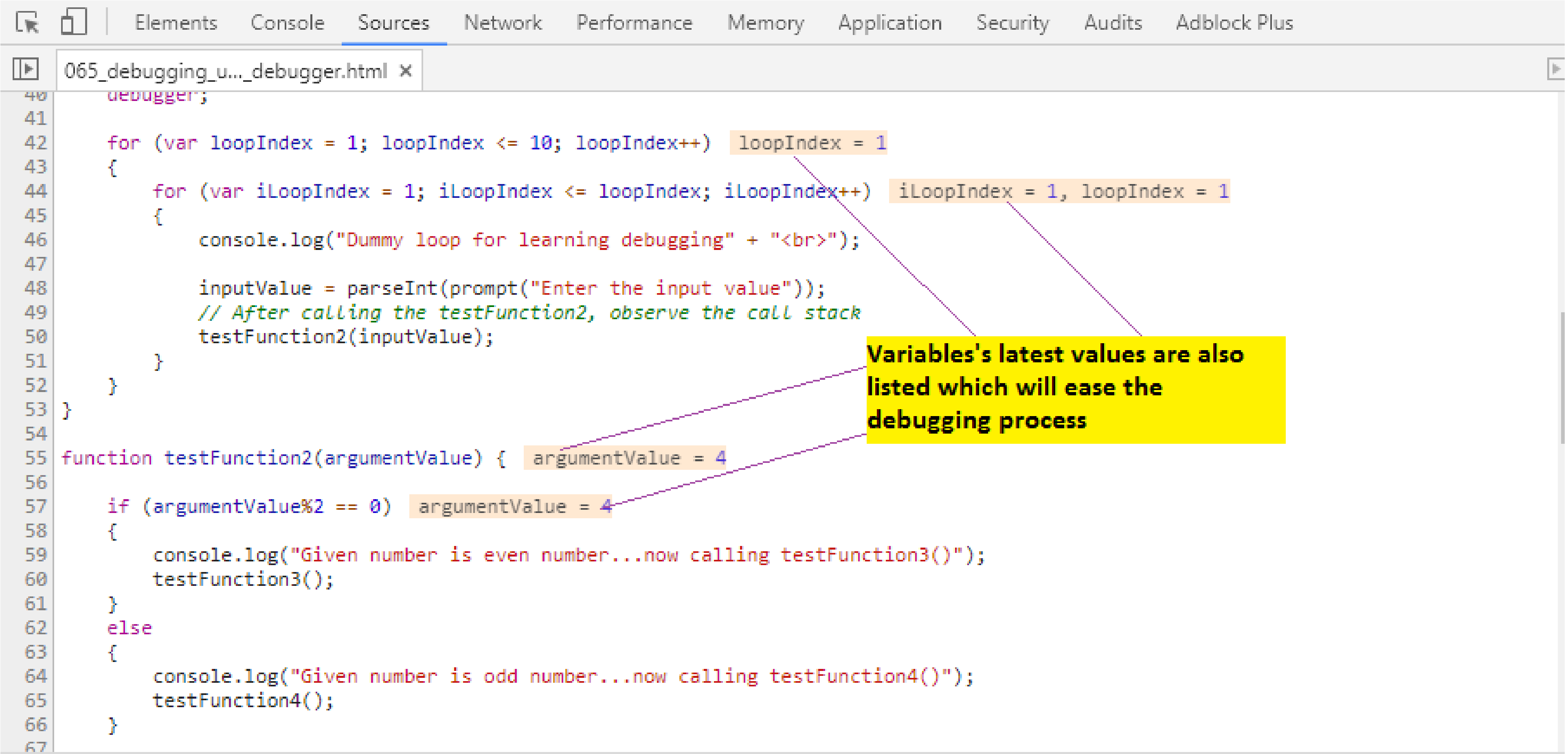
When the debugger is enabled

YOUR cruize control. Navigate them carefully

Call stack - Functions are placed in a LIFO scheme

Variable values - where developers do major mistakes

The Debugger window



```
    debugger;

    for (var loopIndex = 1; loopIndex <= 10; loopIndex++)  loopIndex = 1
    {
        for (var iLoopIndex = 1; iLoopIndex <= loopIndex; iLoopIndex++)  iLoopIndex = 1, loopIndex = 1
        {
            console.log("Dummy loop for learning debugging" + "<br>");

            inputValue = parseInt(prompt("Enter the input value"));
            // After calling the testFunction2, observe the call stack
            testFunction2(inputValue);
        }
    }
}

function testFunction2(argumentValue) {  argumentValue = 4

    if (argumentValue%2 == 0)  argumentValue = 4
    {
        console.log("Given number is even number...now calling testFunction3()");
        testFunction3();
    }
    else
    {
        console.log("Given number is odd number...now calling testFunction4()");
        testFunction4();
    }
```

Variables's latest values are also listed which will ease the debugging process

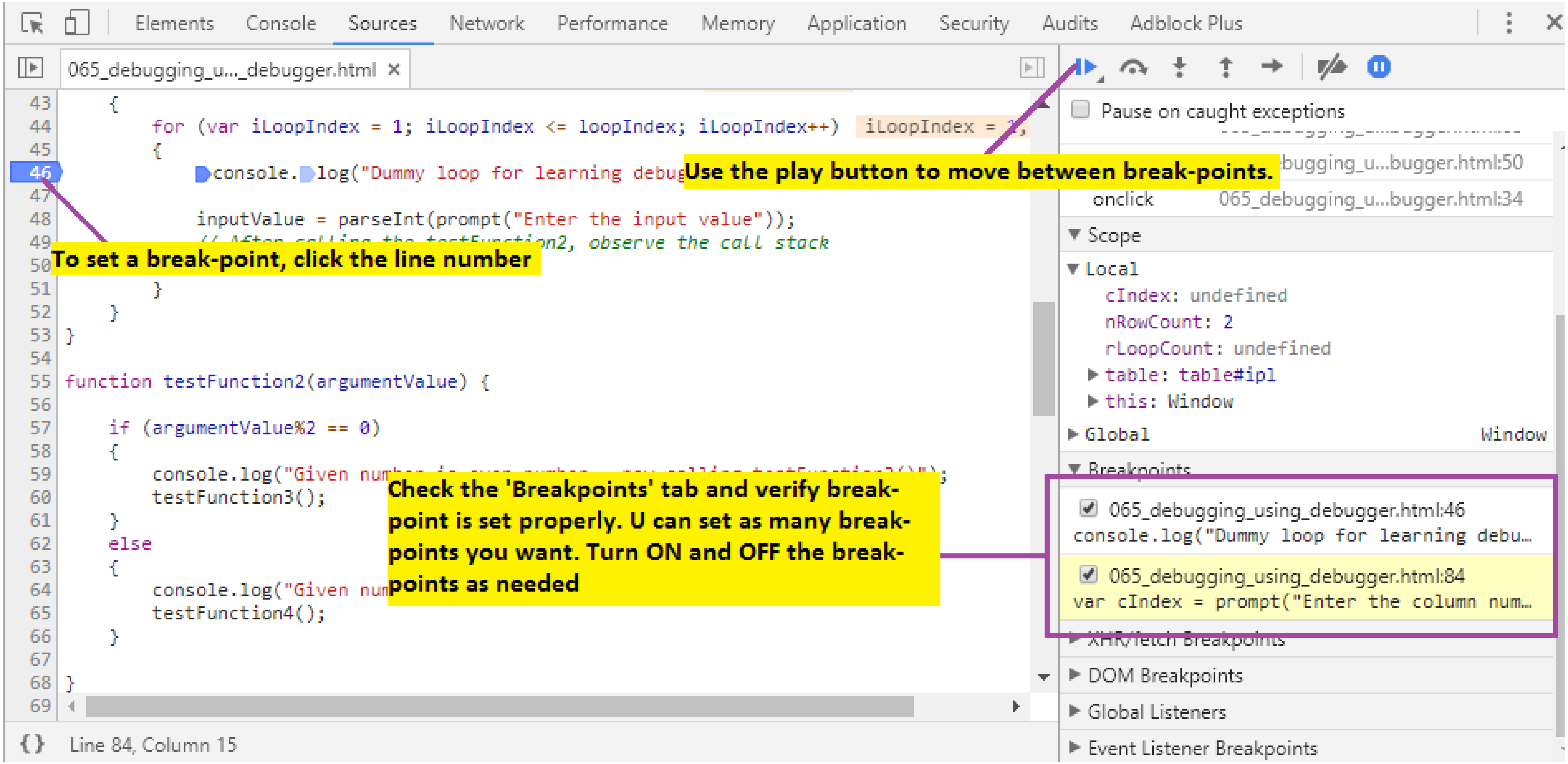
Exercise

- Check the given program and do the following:
  - Various facilities provided
  - Run-time walk through of the code
  - Understand step-in and step-through options and differences between them
  - Understand various segments of a running program (Code, Data, Stack, Heap)
  - What is the call-stack and how it plays a role in function handling?

Breakpoints

- Break Points can be used to stop the execution of the code. They can be set directly in the debugger without using ‘debugger’ keyword
- Multiple break-points can be set at required placed to monitor the code flow. It will help you to investigate the source code at various locations
- You can resume the execution of code by pressing the ‘play’ button, it will further run or pause in the next break-point
- Depending on the issue, breakpoints will help you to narrow down to the code area where the potential problem is there. Upon careful investigation fixes can be done

The Debugger window



```
43     {
44         for (var iLoopIndex = 1; iLoopIndex <= loopIndex; iLoopIndex++)  iLoopIndex = 1,
45         {
46             console.log("Dummy loop for learning debug
47
48             inputValue = parseInt(prompt("Enter the input value"));
49             // After calling the testFunction2, observe the call stack
50
51         }
52     }
53 }
54
55 function testFunction2(argumentValue) {
56
57     if (argumentValue%2 == 0)
58     {
59         console.log("Given num                                          ;
60         testFunction3();
61     }
62     else
63     {
64         console.log("Given num
65         testFunction4();
66     }
67
68 }
69
```

{ } Line 84, Column 15

□ Pause on caught exceptions

	...bugging_u...bugger.html:50
onclick	065_debugging_u...bugger.html:34

▼ Scope

▼ Local

```
cIndex: undefined
nRowCount: 2
rLoopCount: undefined
▶ table: table#ipl
▶ this: Window
```

▶ Global   Window

▼ Breakpoints

☑ 065_debugging_using_debugger.html:46
console.log("Dummy loop for learning debu…

☑ 065_debugging_using_debugger.html:84
var cIndex = prompt("Enter the column num…

▶ XHR/fetch Breakpoints

▶ DOM Breakpoints

▶ Global Listeners

▶ Event Listener Breakpoints

Use the play button to move between break-points.

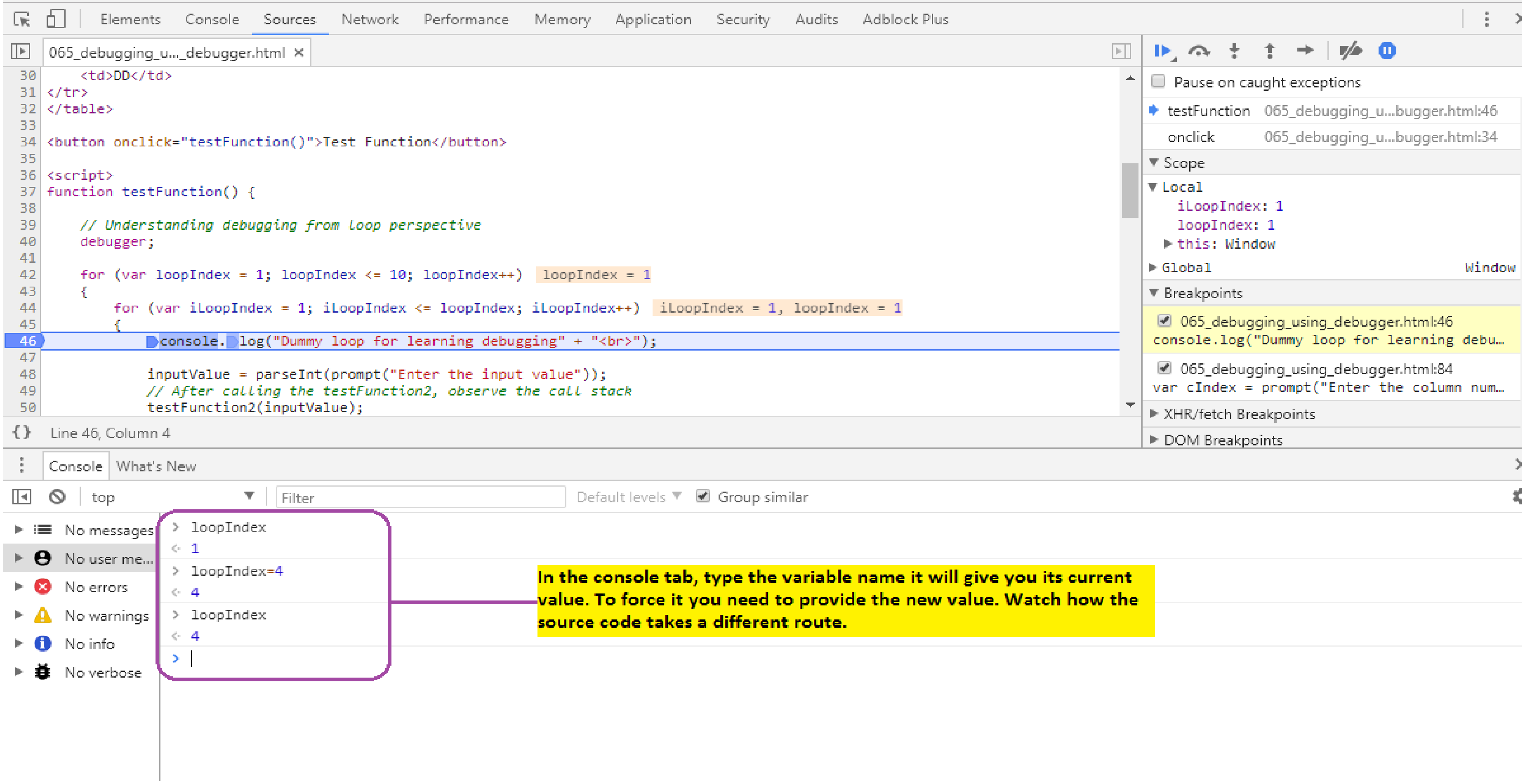
To set a break-point, click the line number

Check the 'Breakpoints' tab and verify break-point is set properly. U can set as many break-points you want. Turn ON and OFF the break-points as needed

Changing variable values in runtime

- There can be some situations during development / testing as follows:
  - Developer not able to test various paths of the code (ex: `if…else`)
  - Developer not able to exactly recreate the problem but knows the path
  - Developer want to force certain conditions and see how the source code handles
- In such situations it would be helpful if the developer is able to force certain variable value during runtime
- This is a crude approach, but helps during the development time. Achieving 100% test coverage using external methods may not be possible all the times
- In Chrome you need to use the ‘console’ tab to set values

The Debugger window



```
    <td>DD</td>
</tr>
</table>

<button onclick="testFunction()">Test Function</button>

<script>
function testFunction() {

    // Understanding debugging from loop perspective
    debugger;

    for (var loopIndex = 1; loopIndex <= 10; loopIndex++)  loopIndex = 1
    {
        for (var iLoopIndex = 1; iLoopIndex <= loopIndex; iLoopIndex++)  iLoopIndex = 1, loopIndex = 1
        {
            console.log("Dummy loop for learning debugging" + "<br>");

            inputValue = parseInt(prompt("Enter the input value"));
            // After calling the testFunction2, observe the call stack
            testFunction2(inputValue);
```

Line 46, Column 4

□ Pause on caught exceptions

testFunction	065_debugging_u...bugger.html:46
onclick	065_debugging_u...bugger.html:34

```
▼ Scope
▼ Local
    iLoopIndex: 1
    loopIndex: 1
  ▶ this: Window
▶ Global                              Window
```

▼ Breakpoints

☑ 065_debugging_using_debugger.html:46
`console.log("Dummy loop for learning debu…`

☑ 065_debugging_using_debugger.html:84
`var cIndex = prompt("Enter the column num…`

▶ XHR/fetch Breakpoints

▶ DOM Breakpoints

```
> loopIndex
< 1
> loopIndex=4
< 4
> loopIndex
< 4
>
```

In the console tab, type the variable name it will give you its current value. To force it you need to provide the new value. Watch how the source code takes a different route.

Exercise

- Check the given program and do the following:
  - Remove debugger statement
  - Setting multiple break-points and use play button to move between
  - Investigate run-time snapshot between multiple break-points
  - Force the code to take a different path in the if…else condition
  - Add / Remove multiple break-points

Avoiding Mistakes

(JavaScript – Strict Mode)

Strict Mode

- Strict mode is declared by adding "use strict"; to the beginning of a script or a function.
- Declared at the beginning of a script, it has global scope.

```
"use strict";

// This will cause an error (x is not declared).

x = 3.14;
```

Strict mode

- Strict mode makes it easier to write "secure" JavaScript.
- Objects are variables too so without declaring object we cannot initialize.
- In strict mode certain operations are not permitted (ex: Deleting an object) are not allowed. This also helps to keep code safe by avoiding some mistakes.

```
"use strict";

 x = {p1:10, p2:20};  // error
```

Thank you

WebStack Academy

#83, Farah Towers,
1st Floor, MG Road,
Bangalore – 560001

M: +91-809 555 7332
E: training@webstackacademy.com

WSA in Social Media:

slideshare