Javascript

Agenda

- ❖ More about functions…
  - ➢ Throwing and catching error.
  - ➢ Callbacks (Introduction)
  - ➢ Self invoking functions.
  - ➢ Closure
  - ➢ Scope
  - ➢ Higher Order functions.
  - ➢ Hoisting
  - ➢ Anonymous / single-use functions

- ❖ Using In-built functions
  - ➢ Functions on array (e.g Array.sort)

Why Error occurred!!!

- Errors can be coding errors made by the programmer
- errors due to wrong input
- other unforeseeable things

Throwing and catching error

try: Test a block of code for errors.
catch: Handle the error.
throw: Create custom errors.
finally: Execute code, after try and catch, regardless of the result.

try and catch

```
try {
    // Block of code to try
}
catch(err) {
   // Block of code to handle errors
}
```

try and catch Example

```
<!DOCTYPE html>
    <html>
        <body>
            <p id="demo"></p>
            <script>
            try {
                showUser("Welcome guest!");
                }
            catch(err) {
                document.getElementById("demo").innerHTML = err.message;
                }
            </script>
        </body>
    </html>
```

try and catch Example

```
<!DOCTYPE html>
    <html>
        <body>
            <p id="demo"></p>
            <script>
            try {
                showUser("Welcome guest!");
               alert("We are doing good");
                }
            catch(err) {
                alert("Ops!! some thing wrong, We are not doing good");

                }
            </script>
        </body>
    </html>
```

throw Statement

The **throw** statement allows you to create a custom error.
The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big";    // throw a text
throw 500;          // throw a number
```

throw Example

```
try {
    if (x == "") throw "empty";
    if (isNaN(x)) throw "not a number";
    x = Number(x);
    if (x < 5) throw "too low";
    if (x > 10) throw "too high";
}
catch (err) {
    message.innerHTML = "Input is " + err;
}
```

finally Statement

The **finally** statement lets you execute code, after try and catch, regardless of the result:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

Exercise Time...

Calculate simple interest
S = (PA * R * T)/100
PA: should be number >100 < 100000
T: Time / Year >=1 and <=5
R: Rate of interest 4 to 12 %

Callback

A callback function, also known as a higher-order function, is a function that is passed to another function (let’s call this other function “otherFunction”) as a parameter, and the callback function is called (or executed) inside the otherFunction.

How to Write a Callback Function

```
function mySandwich(param1, param2, callback) {
    alert('Started eating my sandwich.\n\nIt has: ' + param1 + ', ' + param2);
    callback();
}
mySandwich('ham', 'cheese', function () {
    alert('Finished eating my sandwich.');
});
```

Callback Example

```
<script>
   function sumFun(a, b) {
       return a +b;
   }

  var sum =  sumFun(2, 3);
  sum = sum * 5;
  console.log(sum);
</script>
```

Callback Example

```
<script>
   function sumFun(a, b) {
      setTimeout(function(){
         return a +b;
      }, 100)
   }

  var sum =  sumFun(2, 3);
  sum = sum * 5;
  console.log(sum);
</script>
```

Exercise Time...

```
console.log('A');
setTimeout(function(){
console.log('B');
}, 1000);
console.log('C');
```

Self invoking functions

A self-invoking anonymous runs automatically/immediately when you create it and has no name, hence called anonymous. Here is the format of self-executing anonymous function:

```
(function(){
   // some code…
})();
```

Self invoking Example

```
<script>
 for(var i = 0; i < 3; i++) {
   setTimeout(function(){
            console.log(i);
        }, 100);
 }
</script>
```

Self invoking Example

```
<script>
 for(var i = 0; i < 3; i++) {
    (function(arg){
       setTimeout(function(){
          console.log(arg);
       }, 100);
    })(i);

 }
</script>
```

Closure

Closures are functions that refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, these functions 'remember' the environment in which they were created.

Closure Contd.

JavaScript variables can belong to the **local** or **global** scope.

Global variables can be made local (private) with **closures**.

Closure Contd.

```
var counter = 0;

function add() {
counter += 1;
}

add();
add();
add();
```

Closure Contd.

```
function add() {
var counter = 0;
counter += 1;
}

add();
add();
add();
```

Closure Contd.

```
function add() {
var counter = 0;
function plus() {counter += 1;}
plus();
return counter;
}
```

Closure Contd.

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

add();
add();
add();
```

Scope

In JavaScript, scope refers to the current context of your code. Understanding JavaScript scope is key to writing bulletproof code and being a better developer.

Global Scope

Before you write a line of JavaScript, you’re in what we call the Global Scope. If we declare a variable, it’s defined globally:

example : -

```
var a = 1;

// global scope
function one() {
  alert(a); // alerts '1'
}
```

Functional Scope

Variables declared within a JavaScript function, become LOCAL to the function.

example :-

```
var a = 1;

function two(a) {
  alert(a); // alerts the given argument, not the global value of '1'
}

// local scope again
function three() {
  var a = 3;
  alert(a); // alerts '3'
}
```

Scope Chain

Scope chains establish the scope for a given function. Each function defined has its own nested scope as we know, and any function defined within another function has a local scope which is linked to the outer function - this link is called the chain. It’s always the position in the code that defines the scope. When resolving a variable, JavaScript starts at the innermost scope and searches outwards until it finds the variable/object/function it was looking for.

Scope Chain example

```
function outermost(){
   var x = 'outermost';
   function intermediate(){
      var  y = 'intermediate';
      function innermost(){ // gets
           var z = 'innermost';
           console.log(x, y, z);
      }
      innermost();
   }
   intermediate();
}

outermost();  // prints    outermost intermediate innermost
```

Hoisting

- All scopes in JavaScript are created with Function Scope only, they aren’t created by for or while loops or expression statements like if or switch.

- All *variable* ***declarations,*** anywhere inside a function are "**hoisted**" to the *top* of the function.

Hoisting

Consider this code snippet:

```
function foo() {
	console.log(a);
	var a = 20;
	console.log(a);
}
```

Hoisting

It will be executed as :

```
function foo() {
        var a;
        console.log(a);
        a = 20;
        console.log(a);
}
```

Hoisting Cont.

- JavaScript only hoists declarations, not initializations. If you are using a variable that is declared and initialized after using it, the value will be undefined. The below two examples demonstrate the same behavior.

Example

```
var x = 1; // Initialize x
console.log(x + " " + y);  // prints 1 undefined
var y = 2;
//the above code and the below code are the same

var x = 1; // Initialize x
var y; // Declare y
console.log(x + " " + y);  //y is undefined
y = 2; // Initialize y
```

Higher order function

A higher-order function is a function that can take another function as an argument, or that returns a function as a result.

Taking Functions as Arguments

Since JavaScript is single-threaded, it allows for asynchronous behavior, so a script can continue executing while waiting for a result. The ability to pass a callback function is critical when dealing with resources that may return a result after an undetermined period of time.

Taking Function as Argument (Cont.)

For example, consider this snippet of simple JavaScript that adds an event listener to a button.

```
<button id="clicker">So Clickable</button>

document.getElementById("clicker").addEventListener("click", function() {
  alert("you triggered " + this.id);
});
```

Returning function as Result

- In addition to taking functions as arguments, JavaScript allows functions to return other functions as a result.

Example

```
var snakify = function(text) {
  return text.replace(/millenials/ig, "Snake People");
};
console.log(snakify("The Millenials are always up to something."));
// The Snake People are always up to something.
```

Anonymous function

- An anonymous function is a function that was declared without any named identifier to refer to it.
- As such, an anonymous function is usually not accessible after its initial creation hence also known as single-use functions

Anonymous function (Contd.)

Normal function definition:

```
function hello() {
  alert('Hello world');
}
hello();
```

Anonymous function definition:

```
var anon = function() {
  alert('I am anonymous');
};
anon();
```

In-built functions

http://hepunx.rl.ac.uk/webtempfiles/adye/public_html/jsspec11/builtin.htm

Thank You