



Javascript

Agenda

- ❖ More about functions...
 - Throwing and catching error.
 - Function(invocation and declaration only)
 - Scope
 - Hoisting
 - Closure
 - Self invoking functions.
 - Callbacks (Introduction)
 - Higher Order functions.
 - 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 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>
```

A decorative geometric pattern at the bottom of the slide, composed of various colored triangles in shades of orange, red, pink, and purple.

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!");
        alert("We are doing good");
      }
      catch(err) {
        alert("Ops!! some thing wrong, We are not doing good");
      }
    </script>
  </body>
</html>
```

A decorative geometric pattern at the bottom of the slide, composed of various colored triangles in shades of orange, red, pink, and purple.

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...

```
let a = 1;  
//let p = 2;  
  
try {  
  a = 1/p;  
  console.log(`New value of 'a' is ${a}`);  
} catch (err) {  
  console.log(err.message);  
} finally {  
  console.log(`Final value of 'a' is ${a}`)  
}
```

Output = ?



Functions - Invoke & Declare

The code inside a JavaScript function will execute when "something" invokes it.

```
function myFunction(a, b) {  
    return a * b;  
}  
myFunction(10, 2);
```

1. The function above does not belong to any object. But in JavaScript there is always a default global object.
2. In HTML the default global object is the HTML page itself, so the function above "belongs" to the HTML page.
3. In a browser the page object is the browser window. The function above automatically becomes a window function.
4. myFunction() and window.myFunction() is the same function:



The this Keyword

1. In JavaScript, the thing called this, is the object that "owns" the current code.
2. The value of this, when used in a function, is the object that "owns" the function.
3. When a function is called without an owner object, the value of this becomes the global object.

```
var x = myFunction();           // x will be the window object
function myFunction() {
  return this;
}
```

```
var myObject = {
  firstName:"John",
  lastName: "Doe",
  fullName: function () {
    return this.firstName + " " + this.lastName;
  }
}
myObject.fullName();
```



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'  
}
```



Exercise Time

```
var a = 5;  
function foo() {  
  var a = 7;  
  alert(a);  
}
```

There are now two variables with the same name, a. Which one does Javascript pick?



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); //1 undefined
y = 2; // Initialize y
```

A decorative geometric pattern at the bottom of the slide, composed of various colored triangles in shades of orange, red, pink, and purple.

Exercise Time

function

foo

```
    {  
    {  
    if(true)  
    var    a    =    5;  
    }  
    alert(a);
```

}



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.

```
function sayHello(name) {  
  var text = 'Hello ' + name;  
  var say = function() { console.log(text); }  
  say();  
}  
sayHello('Joe');
```



Closure Contd.

```
function sayHello2(name) {  
  var text = 'Hello ' + name; // Local variable  
  var say = function() { console.log(text); }  
  return say;  
}  
var say2 = sayHello2('Bob');  
say2(); // logs "Hello Bob"
```




Closure Contd.

```
function say667() {  
  // Local variable that ends up within closure  
  var num = 42;  
  var say = function() { console.log(num); }  
  num++;  
  return say;  
}  
var sayNumber = say667();  
sayNumber(); // logs 43
```



Closure Contd.

```
var gLogNumber, gIncreaseNumber, gSetNumber;  
function setupSomeGlobals() {  
  // Local variable that ends up within closure  
  var num = 42;  
  // Store some references to functions as global variables  
  gLogNumber = function() { console.log(num); }  
  gIncreaseNumber = function() { num++; }  
  gSetNumber = function(x) { num = x; }  
}
```

A decorative geometric pattern at the bottom of the slide, consisting of various colored triangles (orange, red, pink, purple, blue) arranged in a jagged, mountain-like shape.

Closure Contd.

```
setupSomeGlobals();  
gIncreaseNumber();  
gLogNumber(); // 43  
gSetNumber(5);  
gLogNumber(); // 5
```

```
var oldLog = gLogNumber;
```

```
setupSomeGlobals();  
gLogNumber(); // 42
```

```
oldLog() // 5
```



Closure Contd.

```
function buildList(list) {  
  var result = [];  
  for (var i = 0; i < list.length; i++) {  
    var item = 'item' + i;  
    result.push( function() {console.log(item + ' ' + list[i])} );  
  }  
  return result;  
}  
  
function testList() {  
  var fnlist = buildList([1,2,3]);  
  // Using j only to help prevent confusion -- could use i.  
  for (var j = 0; j < fnlist.length; j++) {  
    fnlist[j]();  
  }  
}
```

testList() //logs "item2 undefined" 3 times



Closure Contd.

```
function sayAlice() {  
  var say = function() { console.log(alice); }  
  // Local variable that ends up within closure  
  var alice = 'Hello Alice';  
  return say;  
}  
sayAlice(); // logs "Hello Alice"
```



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

Exercise Time

```
var a = 6;
function test() {
  var a = 7;
  function again() {
    var a = 8;
    alert(a);
  }
  again();
  alert(a);
}
test();
alert(a);
```

When executed, this will pop up three alerts. In order, what are they and what will be the value displayed in it ?

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>
```



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');
```

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", "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

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array



Array Methods

- **join**: Joins array elements using provided separator and returns string.
- **reverse**: Reverse the order of elements in the array.
- **sort**: Sorts array elements in place using default sorting for elements.
- **concat**: Creates and return new array that contains concatenation of arrays on which it is invoked
- **slice**: Returns a slice(subarray) of the array.
- **splice**: General purpose method for insertion and removal.
- **push**: Appends one or more elements at the end of the array.
- **pop**: Deletes the last element of the array, and returns the element.
- **shift**: Removes and return first element of array
- **unshift**: Adds one or more elements to the beginning of the array.
- **indexOf**: Returns the index of specified element in the array, otherwise returns -1.



Array Methods

- **forEach**: Provides general iteration over arrays
- **map**: Used for transformation purposes, maps each element of array to a function for desired output
- **filter**: Used to filter array based on a predicate
- **reduce**: Combine array elements to reduce to a single value.
- **every**: Checks if every array element satisfies the given predicate.
- **some**: Checks if some(one or more) elements satisfy the given predicate.



Exercise time

- 1)

```
var city = new Array("delhi", "agra", "akot", "aligarh","palampur");  
console.log(city.slice(2));
```
- 2)

```
var city = new Array("delhi", "agra", "akot", "aligarh","palampur");  
console.log(city.shift());
```
- 3)

```
var city = new Array("delhi", "agra", "akot", "aligarh","palampur");  
console.log(city.pop());
```
- 4)

```
var city = ["delhi", "agra", "akot", "aligarh"];  
console.log(city.sort());
```



Brain teaser

```
(function() {  
  var a = [3,5,7,8,7,2];  
  var b = a.filter(function(num){  
    return num%2;  
  });  
  console.log(b);  
})();
```

API for Assignment

"https://maps.googleapis.com/maps/api/geocode/json?latlng=40.714224,-73.961452"



Thank You

