let scope

1. let scope is similar to var.

```
function letDemo(){
      let age=25;
      if(new Date().getFullYear()==2017){
            age=32;
            console.log(age);
      }
      console.log(age);
}
```

2. Unlike var, you cannot define the same let variable more than once inside a block:

```
function letDemo(){
      let myage = 39
      let myname = 'Capgemini'
      let myage = 40 // SyntaxError: Identifier myage has
already been declared
}
```

Note :this will show error myage already declared.

3. let is block scoped, which just means it's available inside the block (curly braces) it's defined in (including inner blocks), but not outside it:

```
function letDemo(){
      if (true){ // new block
          let myname = 'Capgemini'
      }
      console.log(myname) // syntax error, myname is
undefined
}
```

4. Defining multiple let variables with the same name

```
function letDemo(){
  let mybrother = 'Paul'
       if (true){ // new block
           let mybrother = 'Jason'
           console.log(mybrother) // Jason
       }
       console.log(mybrother) // Paul
  }
```

5. Using let inside for(...) loops

```
function greet(){
     alert('hello');
}

function letDemo(){
     /*for (var i = 0; i < 5; i++){
         setTimeout(function(){
             console.log(i)
         }, i * 100)
     }
     //logs '5, 5, 5, 5, 5'
*/
     /*for (var i = 0; i < 5; i++){
         (function(x){
             setTimeout(function(){
                 console.log(x)
             }, i * 100)
         })(i)
     }
     //logs '0, 1, 2, 3, 4'
     //this is IIFE inside
*/
     for (let i = 0; i < 5; i++){
         setTimeout(function(){
```

```
            console.log(i)
        }, i * 100)
    }
    //logs '0, 1, 2, 3, 4'
}
```

Note: *This is helpful in many scenarios, such as Ajax requests inside loops that rely on the value of i to make different requests, or event handlers that utilize the value of i to perform an action tailored to the element it's bound to.*

Example for Let:

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>MyApplication Demo</title>
<script type="text/javascript" src="script/demo.js"></script>
</head>
<body>

<a href="#" onclick="letExample()">Home</a><br>
<a href="#" onclick="letExample()">Employee </a><br>
<a href="#" onclick="letExample()">Customer</a><br>
<a href="#" onclick="letExample()">Feedbacks</a><br>
<a href="#" onclick="letExample()">Contact Us</a><br>
<a href="#" onclick="letExample()">Click here</a><br>

</body>
</html>
```

demo.js

```
function letExample(){
    let links = document.getElementsByTagName('a')
    for (let i=0; i<links.length; i++){
        links[i].onclick = function(){
```

```
            alert('You clicked on link ' + (i+1))
        }
    }
}
```

6. let variables and the Temporal Dead Zone

It is a term in the programming world called the Temporal Dead Zone, and it has nothing to do with zombies or time shifting, sadly.

```
function test(){
    console.log(dog) // returns ReferenceError
    let dog = 'spotty'
}


function test(){
    console.log(dog) // returns undefined
    var dog = 'spotty' // variable auto hoisted to the
very top of the function (but not its value)
    console.log(dog) // ‘spotty’
}
```

const scope

1. const variable must be initialized

```
function test(){
      const mydog = 'spotty' // good
            mydog = 'fluffy' // error: assignment to constant

      const mybrother; // error: Missing initializer in
const
}
```

2. While a const variable cannot be reassigned entirely to a different value, if the value of a const is an object or array, the object's properties themselves are still mutable, able to be modified:

```
function test(){
      const myobject = {name:'George', age:39}
      //myobject = {name: 'Ken', age:39} //error
      myobject.age = 40 // OK

      const myarray = []
      myarray[0] = 'Football' // OK
}
```

Arrow function

- JavaScript arrow functions are anonymous functions
- Arrow functions cannot be used as constructor functions, (ie: with the keyword new)
- Lexical binding of this inside arrow function: The value of this inside an arrow function always points to the same this object of the scope the function is defined in, and never changes.

Arrow function syntax

```
//single param, single statement
x => x;
```

Regular function equivalent

```
function(x){
    return x;
}
```

Arrow function syntax

```
//single param, single statement
x => x + 5;
```

Regular function equivalent

```
function(x){
    return x + 5;
}
```

Arrow function syntax

```
//multiple params
(x, y) => x * y;
```

Regular function equivalent

```
function(x, y){
    return x * y;
}
```

If the function BODY contains more than one statement, wrap them in curly braces:

Arrow function syntax

```
//multiple params and statements
(x, y) => {
    var factor = 5;
    var growth = (x-y) * factor;
}
```

Regular function equivalent

```
function(x, y){
    var factor = 5;
    var growth = (x-y) * factor;
}
```

Arrow function syntax

```
//Single object literal statement
//Wrap function BODY in parenthesis
x => ({id: x, height: 270});
```

```
// OR
// Use an explicit return
x => {return {id: x, height: 270}};
```

Regular function equivalent

```
function(x){
    return {id: x, height: 270};
}
```

The **syntax for JavaScript arrow functions** is very easy to remember with the following rules:

- For arrow functions with a single parameter, you can omit the parenthesis () around the parameter.
- For arrow functions with no or multiple parameters, wrap the parameters in parenthesis.
- For arrow functions with a single BODY statement, you can omit the braces {} around the statement. The value derived from the statement is automatically returned with no braces.
- For arrow functions with multiple BODY statements, wrap them in curly braces. No value is automatically returned with braces- use the return statement to specify the value.
- If a function BODY contains only a single object literal, wrap the function BODY in parenthesis () to differentiate it from the object literal wrapper.

Promises

Syntax

```
var mypromise = new Promise(function(resolve, reject){
 // asynchronous code to run here
 // call resolve() to indicate task successfully completed
 // call reject() to indicate task has failed
})
```

```
function getImage(url){
    return new Promise(function(resolve, reject){
        var img = new Image()
        img.onload = function(){
            resolve(url)
        }
        img.onerror = function(){
            reject(url)
        }
        img.src = url
       // alert(img.src);
        //document.getElementById('img1').src=url;
    })
}
```

The then() and catch() methods

Whenever you instantiate a Promise object, two methods- **then() and catch()-** become available to decide what happens next after the conclusion of an asynchronous task. Take a look at the below:

```
getImage('doggy.jpg').then(function(successurl){

    document.getElementById('doggyplayground').innerHTML = '<img src="' +
successurl + '" />'
})
```

Example

```
function testPromise(){
     getImage('micky1.png').then(
               function(successurl){
                   document.getElementById('cnt').innerHTML = '<img
src="' + successurl + '" />'
               },
               function(errorurl){
                   console.log('Error loading ' + errorurl)
               }
          )
}


function getImage(url){
    return new Promise(function(resolve, reject){
        var img = new Image()
        img.onload = function(){
            resolve(url)
        }
        img.onerror = function(){
            reject(url)
        }
        img.src = url
       // alert(img.src);
        //document.getElementById('img1').src=url;
    })
}
```

```
<!DOCTYPE html>
<html>
<head>
```

```
<meta charset="ISO-8859-1">
<title>MyApplication Demo</title>
<script type="text/javascript"
src="script/demo.js"></script>
</head>
<body onload="testPromise()">
<div id="cnt">

</div>
</body>
</html>
```

Access with catch

```
function testPromise(){
      /*getImage('micky1.png').then(
                function(successurl){
                    document.getElementById('cnt').innerHTML = '<img
src="' + successurl + '" />'
                },
                function(errorurl){
                    console.log('Error loading ' + errorurl)
                }
            )*/

      getImage('../micky1.png').then(function(successurl){
          document.getElementById('cnt').innerHTML = '<img src="' +
successurl + '" />'
      }).catch(function(errorurl){
          console.log('Error loading ' + errorurl)
      })
}
```

Calling catch() is equivalent to calling then(undefined, function), so the above is the same as:

```
getImage('doggy.jpg').then(function(successurl){
    document.getElementById('doggyplayground').innerHTML = '<img src="' +
successurl + '" />'
}).then(undefined, function(errorurl){
    console.log('Error loading ' + errorurl)
})
```

Example

```
<!DOCTYPE html>
<html>
<head>
<meta charset="ISO-8859-1">
<title>Insert title here</title>
<script type="text/javascript"
src="script/promiseDemo.js"></script>
<style type="text/css">
.demoarea{
border:2px solid darkbrown;
padding:5px;
}

.demoarea img{
margin-right: 10px;
}

</style>
</head>
<body >
<div id="doggyplayground" class="demoarea">


</div>
```

```
<button onclick="demo1()">show</button>
</body>
</html>
```

promiseDemo.js

```
function displayimages(images){
     var doggyplayground =
document.getElementById('doggyplayground');

    var targetimage = images.shift(); // process doggies images
one at a time
    if (targetimage){ // if not end of array
        getImage(targetimage).then(function(url){ // load image
then...
            var dog = document.createElement('img')
            dog.setAttribute('src', url)
            doggyplayground.appendChild(dog) // add image to DIV
            displayimages(images) // recursion- call
displayimages() again to process next image/doggy
        }).catch(function(url){ // handle an image not loading
            console.log('Error loading ' + url)
            displayimages(images) // recursion- call
displayimages() again to process next image/doggy
        })
    }
}

function getImage(url){
    return new Promise(function(resolve, reject){
        var img = new Image()
        img.onload = function(){
            resolve(url)
        }
        img.onerror = function(){
            reject(url)
```

```
        }
        img.src = url
       // alert(img.src);
        //document.getElementById('img1').src=url;
    })
}

function demo1(){
     var doggies = ['dog1.png', 'dog2.png', 'dog3.png',
'dog4.png', 'dog5.png']
     //doggyplayground.innerHTML = ''
     displayimages(doggies)
}
```

Async Functions

The Anatomy of an Async Function

Async functions are defined by placing the async keyword in front of a function, such as:

```
async fetchdata(url){
    // Do something
    // Always returns a promise
}
```

This does two things:

- It causes the function to always return a promise whether or not you explicitly return something, so you can call then() on it for example. More on this later.
- It allows you to use the await keyword inside it to wait on a promise until it is resolved before continuing on to the next line inside the async function.

"Async functions always returns a promise. If we explicitly return a value at the end of our async function, the promise will be resolved with that value; otherwise, it resolves with undefined."

Closure

An Example of a Closure

Two one sentence summaries:

- a closure is the local variables for a function - kept alive *after* the function has returned, or
- a closure is a stack-frame which is *not deallocated* when the function returns. (as if a 'stack-frame' were malloc'ed instead of being on the stack!)

```
function sayHello(name){
    var text = 'Hello ' + name;
    var sayAlert = function(){ alert(text); }
    return sayAlert();
}
```

Callback function

```
var myCallBackExample = {
   myFirstFunction : function( param1, param2, callback ) {
      // Do something with param1 and param2.
      if ( arguments.length == 3 ) {
         // Execute callback function.
         // What is the "best" way to do this?
      }
   },
   mySecondFunction : function() {
      myFirstFunction( false, true, function() {
         // When this anonymous function is called, execute it.
      });
   }
};
```

Prototype

The JavaScript prototype property allows you to add new properties to an existing prototype

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

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

Note:

Only modify your **own** prototypes. Never modify the prototypes of standard JavaScript objects.