ES6 / ES2015

What is ES6

- The most recent version ECMAScript / javascript
- First major change since ES5 (2009)
- ES6 and ES2015 are same
- Modern syntax
- New features & standard library
- Introduce classes

Compatibility

- Some browsers may still not support ES6
- Latest Chrome and Firefox do support
- Transpilers used to convert ES6 to ES5
  - Babel
  - Traceur
  - Closure

What is new in ES6

- let and const declarations
- New data structures
- Iterators
- Arrow functions
- Classes and inheritance
- Template strings
- String features
- Math & Number features
- Promises
- Generators

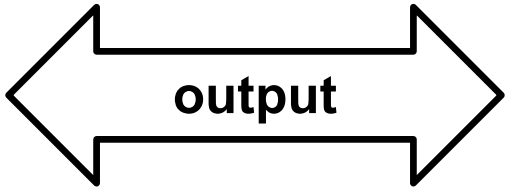
let

- let is same as var in global scope
- var changes the variable of global scope
- let allows to declare local variables

```
function varTest(){
    var a =30;
      if(true){
            var a = 50;
            console.log(a);
      }
      console.log(a);
}
```

```
function letTest(){
    let a =30;
      if(true){
            let a = 50;
            console.log(a);
      }
      console.log(a);
}
```

50
50



50
30

const

- Used to declare constants

```
const colors=[ ];
colors.push('red');
colors.push('blue'):

console.log(colors);

colors =  345;       // error
```

for of loop

```
nums=[23,45,65,44,77];
console.log(nums)

for(let n of nums)
   console.log(n)
```

classes

- Classes used similar to java

```
class User{
    constructor(name,address){
        this.name=name;
        this.address=address;
     }
     show(){
        console.log(this.name+" "+this.address+" "+this.salary);
     }

}
```

```
let user1 = new User("Ramana","Hyderabad");
user1.show();
```

classes - static members

- Classes can have static variables and functions
- instance methods refer static members with class name
- this inside a static method refers static data

```
class User {
   constructor(name,address){
      this.name=name;
      this.address=address;
      User.count++;
   }
   static count=0;
   static showCount(){
      console.log(this.count);
   }
}
```

```
let user1 = new User("Ramana","Hyderabad");
let user2 = new User("Siddarth","Delhi");
user1.show();
user2.show();
User.showCount();
```

classes - inheritance

- Classes can extend other classes and add new data and methods
- instance methods refer static members with class name
- this inside a static method refers static data

```
class  Employee extends  User {
   constructor(name,address,salary){
         super(name,address);
         this.salary=salary;
   }
    show(){
       super.show();
      console.log(this.salary);
   }
}
```

```
let emp= new Emp('Ramana','Hyd',5000);
emp.show();
```

String Template

- Using ` string can be extended to multiple lines
- Variables can be embedded into the template

```
let template =  `<body>
                <p>Hello world</p>
                </body>`;

let name = 'Ramana';
function convert(nm){
      return nm.toUpperCase();
}

let message = `Your name is ${name}` ;
let message2=`converted name is ${convert(name)}` ;
```

Object Manipulation

- Spread operator to merge objects
- **ES5**

```
const obj1 = { a: 1, b: 2 }
const obj2 = { a: 2, c: 3, d: 4}
const obj3 = Object.assign( obj1, obj2)
```

- **ES6**

```
const obj1 = { a: 1, b: 2 }
const obj2 = { a: 2, c: 3, d: 4}
const obj3 = {...obj1, ...obj2}
```

obj3 will be → { a: 2, b: 2, c: 3, d: 4 }

Object Manipulation

- Extract multiple values from object

- **ES5**

```
var obj1 = { a: 1, b: 2, c: 3}
var a = obj1.a
var b = obj1.b
var c = obj1.c
```

- **ES6 (object destructuring)**

```
const obj1 = { a: 1, b: 2, c: 3, d: 4 }
let {  a,   b,  c  } = obj1
```

Object Manipulation

- Combining multiple variables into an object

- **ES5**

```
var a = 1
var b = 2
var c = 3
var d = 4
var obj1 = { a: a, b: b, c: c, d: d }
```

- **ES6**

```
var a = 1
var b = 2
var c = 3
var d = 4
var obj1 = { a, b, c, d }
```

Object Manipulation

- Changing variable names

```
let obj={firstName:'Ramana',lastName:'Reddy',address:'Hyderabad'};

let {firstName:fn,lastName:ln}=obj
console.log( `${fn} ${ln}`)
```

Array destructuring

- Extracting array elements

```
let nums=[12,10,56,33];

let [a, …b]=nums;
console.log(b)
```

b will be → [10, 56, 33]

- Extracting array elements by selection

```
let nums=[12,10,56,33];

let [a, , , b]=nums;
console.log(b)
```

b will be → 33

Arrow functions

- Arrow function brings clarity and code reduction

```
function greetings (name) {
   return 'hello ' + name
}
```

Can be written as any of these

```
const  greetings =  (name) =>  `hello ${ name}` ;
const greetings = name =>   `hello ${name}`;
const greetings = name => {   return `hello ${name}` ; }  ;
```

Promises

- Promises give us a way to handle asynchronous processing in a more synchronous fashion
- They represent a value that we can handle at some point in the future
- Promises give us guarantees about that future value
- The standard way to create a Promise is by using the new Promise constructor which accepts a handler that is given two functions as parameters.
- The first handler (*typically named* resolve) is a function to call with the future value when it's ready
- the second handler (*typically named* reject) is a function to call to reject the Promise if it can't resolve the future value

Creating promise

```
function divide(a, b) {
    let promise = new Promise((resolve, reject) =>   {
            if (b != 0)
                resolve(a / b);
            else
                reject("Zero divide error");
        }
      );

    return promise;
}
```

Consuming a Promise

- To consume the Promise attach a handler to the Promise using it's .then() method. This method takes a function that will be passed the resolved value of the Promise once it is fulfilled.

```
divide(30,10).then(  result=> { console.log(result); }    );
```

- Promise's **then()** method actually takes two possible parameters. The first is the function to be called when the Promise is fulfilled and the second is a function to be called if the Promise is rejected

```
divide(30,0)
        .then(
                result=>{console.log(result);},
                error=> { console.log(error);}
    );
```

Consuming a Promise

- Promise's catch() method can be used to handle error

```
divide(30,0)
        .then(result=>{console.log(result);}  )
        .catch( error=> { console.log(error);  );
```

- Promise's then() method can be chained to other then() method which takes return value of previous then() method

```
divide(30,0)
        .then( result=>  result+100)
         .then( finalresult=> { console.log(finalresult);});
```

Export / Import

- The **export** statement is used when creating JavaScript modules to export functions, objects, or primitive values from the module so they can be used by other programs with the import statement
- There are two different types of export, **named** and **default**
- You can have multiple named exports per module but only one default export
- During the import, it is mandatory to use the same name of the corresponding object with named export
- But a default export can be imported with any name
- it is not possible to use **var, let or const** with export default

Named export

```
// module "my-module.js"
 function cube(x) {
   return x * x * x;
 }
 const foo = Math.PI + Math.SQRT2;

 export { cube, foo };
```

```
// in another module
 import { cube, foo } from './my-module';
 import  { cube } from './my-module';
```

Named export

```
// module "my-module.js"
 export  function cube(x) {
   return x * x * x;
 }

 export  const foo = Math.PI + Math.SQRT2;
```

```
// in another module
 import { cube, foo } from './my-module';
 import  { cube } from './my-module';
```

Default export

```
// module "my-module.js"
 function cube(x) {
   return x * x * x;
 }

 export   default  cube;
```

```
// in another module  all the below imports are OK
 import    cube  from './my-module';
 import   getCube from './my-module';
```

Misc

- New string functions – startswith(), endsWith(), includes()
- numbers - ox33, ob010101, 0345
- New structures – Set, Map, WeakSet, WeakMap with handling methods

- Iterators and Iterables
- Generators