

# ES6

## I. Basic

ECMAScript (ES) is a scripting language specification standardized by ECMAScript International. It is used by applications to enable client-side scripting.

Strict Mode imposes a layer of constraint on JavaScript. It makes several changes to normal JavaScript semantics.

- Example: "use strict"

A = "Strict Mode script"

### Let

let is similar to var but let has scope. let is only accessible in the block level it is defined.

Example: 

```
if (true) {
    let a = 40;
    console.log(a); //40
}
console.log(a); //
undefined
```

### var vs let

```
var x = 40;
var x =49;
console.log(x);    // 49
```

```
let x = 40;
let x = 49;
console.log(x);    // error x
has already been declared
```

### Const

The **const** declaration creates a read-only reference to a value. It does not mean the value it holds is immutable, just that the variable identifier cannot be reassigned. Const has a fix value.

Example: 

```
const x = 10;
x = 56; // show error
```

## II. Loops

+ **For ... in** is used to loop through an object's properties.

Example: 

```
var obj = {a:4, b:6, c:8, d:10};
for (var p in obj){
    console.log(obj[p]);
}
```

It returns index of array or object's properties.

+ **For ... of** is used to iterates through list of elements (i.e) like Array and returns the elements (not their index) one by one.

Example: 

```
let arr = [2,3,4,1];
for (let value of arr) {
```

```
    console.log(value);
}
```

It returns each element of array or object's properties.

### + Label

A **label** is simply an identifier followed by a colon (:) that is applied to a statement or a block of code. A label can be used with **break** and **continue** to control the flow more precisely.

## III. Function

+ **Rest parameters** act as placeholders for multiple arguments of the same type.

Example: 

```
function func(...params){
    console.log(params.length);
}
func(); // 0
func(3, 4, 23, 34); // 4
```

+ **Anonymous Function** is a functions that are not bound to an identifier (function name).

Example:

```
var func = function(x,y){return x + y};
function item() {
    var result;
    result = func(46,74);
    console.log("item result : " + result);
}
item();
```

+ **Lambda functions** are a concise mechanism to represent anonymous functions. These functions are also called as **Arrow functions**. There are 3 parts to a Lambda function

- **Parameters** - A function may optionally have parameters.
- **The fat arrow notation/lambda notation (=>)**: It is also called as the goes to operator.
- **Statements** – Represents the function's instruction set.

Example:

```
let Func = (a, b = 10) => {
    let t = a * 10;
    t -= b;
    return t;
}
console.log(Func(4));
```

+ **Immediately Invoked Function Expressions (IIFEs)** can be used to avoid variable hoisting from within blocks. It allows public access to methods while retaining privacy for variables defined within the function. This pattern is called as a self-executing anonymous function.

Example:

```
var main = function() {
    var loop = function() {
        for(var x = 0;x<5;x++) {
            console.log(x);
        }
    }();
    console.log("x can not be accessed outside the block scope x value is :"+x);
}
main();
```

### + Generator Function

A generator is like a regular function except that –

- The function can yield control back to the caller at any point.
- When you call a generator, it doesn't run right away. Instead, you get back an iterator. The function runs as you call the iterator's next method.

Generators are denoted by suffixing the function keyword with an asterisk; otherwise, their syntax is identical to regular functions. Generators enable two-way communication between the caller and the called function. This is accomplished by using the **yield** keyword. **Note** – Generator functions cannot be represented using arrow functions.

Example:

```
function* ask() {
    const name = yield "What is your name?";
    const sport = yield "What is your favorite sport?";
    return `${name}'s favorite sport is ${sport}`;
}
const it = ask();
console.log(it.next());
```

```
console.log(it.next('Seyha'));
console.log(it.next('Football'));
```

## IV. Object

An **object** is an instance which contains a set of key value pairs. Object can represent multiple or complex values and can change over their life time. The values can be scalar values or functions or even array of other objects.

Example:

```
var person = {
  firstname:"Tom",
  lastname:"Hanks",
  func:function(){return "Hello!!"},
};
```

### - Object Constructor

Example: var Person = new Object();

```
Person.name = "Seyha";
//define an object
```

### - Function Constructor

Example: function Person() {

```
  this.name = "Seyha"
}
```

```
var obj = new Person();
console.log(obj.name);
```

### - The Object.create Method

Example: var roles = {

```
  type: "Admin",
  displayType :
  function() {
    console.log(this.type);
```

```
  }
}
var super_role =
Object.create(roles);
super_role.displayType();
// Output:Admin
var guest_role =
Object.create(roles);
guest_role.type = "Guest";
guest_role.displayType(); //
Output:Guest
```

### - Object De-structuring

The term **destructuring** refers to breaking up the structure of an entity.

Example: var emp = { name: 'John', Id: 3 }

```
var {name, Id} = emp;
console.log(name);
console.log(Id);
```

## V. Array

### - Array.prototype.find

**find** lets you iterate through an array and get the first element back that causes the given callback function to return true. Once an element has been found, the function immediately returns. **Note** – The ES5 **filter()** and the ES6 **find()** are not synonymous. Filter always returns an array of matches (and will return multiple matches), find always returns the actual element.

Example: var numbers = [8, 2, 3, 5];

```
var oddNumber =
numbers.find((x) => x % 2 ==
1);
console.log(oddNumber); // 3
```

### - Array.prototype.findIndex

**findIndex** behaves similar to find, but instead of returning the element that matched, it returns the index of that element.

Example: var numbers = [8, 2, 3, 5];

```
var oddNumber =
numbers.findIndex((x) => x % 2
== 1);
console.log(oddNumber); // 1
```

### - Array.prototype.entries

Entries will return an array of arrays, where each child array is an array of [index, value].

Example: var numbers = [2, 1, 4, 6];

```
var val= numbers.entries();
console.log([...val]);
```

### - Array.from

**Array.from()** enables the creation of a new array from an array like object. The basic functionality of Array.from() is to convert two kinds of values to Arrays –

- Array-like values.

- Iterable values like Set and Map

Example: 

```
for (let i of
Array.from('Seyha')) {
  console.log(i)
}
```

- [Array.prototype.key](#) this function returns the array indexes.

Example: 

```
console.log(Array.from(['a',
'b'].keys()));
```

#### - [Array De-structuring](#)

Example: 

```
var arr = [12,13]
var [x,y] = arr
console.log(x) ;
console.log(y) ;
```

## VI. Collection

### - [Maps](#)

The Map object is a simple key/value pair. Keys and values in a map may be primitive or objects. The set() function sets the value for the key in the Map object. This function returns the Map object. The has() function returns a boolean value indicating whether the specified key is found in the Map object. This function takes a key as parameter.

**Note** – Maps distinguish between similar values but bear different data types. In other words, an **integer key 1** is considered different from a **string key “1”**. Consider the following example to better understand this concept.

Example: 

```
var map = new Map();
map.set(1,true);
console.log(map.has("1"));
//false

map.set("1",true);
console.log(map.has("1"));
//true
```

The **get()** function is used to retrieve the value corresponding to the specified key.

The set() replaces the value for the key, if it already exists in the map. Consider the following example.

Example: 

```
var roles = new Map([
  ['r1', 'User'],
  ['r2', 'Guest'],
  ['r3', 'Admin'],
]);
console.log(`${roles.get('r1')}`)
roles.set('r1', 'User1')
```

```
console.log(roles.get('r1'));
```

### - [Weak Maps](#)

A weak map is identical to a map with the following exceptions –

- Its keys must be objects.
- Keys in a weak map can be Garbage collected. **Garbage collection** is a process of clearing the memory occupied by unreferenced objects in a program.
- A weak map cannot be iterated or cleared.








Example: 

```
let weakMap = new WeakMap();
let obj = {};
console.log(weakMap.set(obj,"hello"));
console.log(weakMap.has(obj)); // true
```

### - [Sets](#)

A set is an ES6 data structure. It is similar to an array with an exception that it cannot contain duplicates. In other words, it lets you store unique values. Sets support both primitive values and object references. Just like maps, sets are also ordered, i.e. elements are iterated in their insertion order.

## - Set Method

Sr.No	Method & Description
1	<b>Set.prototype.add(value)</b>  Appends a new element with the given value to the Set object. Returns the Set object.
2	<b>Set.prototype.clear()</b>  Removes all the elements from the Set object.
3	<b>Set.prototype.delete(value)</b>  Removes the element associated to the value.
4	<b>Set.prototype.entries()</b>  Returns a new Iterator object that contains <b>an array of</b> [value, value] for each element in the Set object, in insertion order. This is kept similar to the Map object, so that each entry has the same value for its key and value here.
5	<b>Set.prototype.forEach(callbackFn[, thisArg])</b>  Calls <b>callbackFn</b> once for each value present in the Set object, in insertion order. If <b>athisArg</b> parameter is provided to forEach, it will be used as the 'this' value for each callback.
6	<b>Set.prototype.has(value)</b>  Returns a boolean asserting whether an element is present with the given value in the Set object or not.
7	<b>Set.prototype.values()</b>  Returns a new Iterator object that contains the <b>values</b> for each element in the Set object in insertion order.

## -Weak Set

Weak sets can only contain objects, and the objects they contain may be garbage collected. Like weak maps, weak sets cannot be iterated.

Example: 'use strict'

```
let weakSet = new WeakSet();
let obj = {msg:"hello"};
weakSet.add(obj);
console.log(weakSet.has(obj));
weakSet.delete(obj);
console.log(weakSet.has(obj));
```

## - Iterator

Iterator is an object which allows to access a collection of objects one at a time. Both set and map have methods which returns an iterator.

Iterators are objects with **next()** method. When next() method is invoked, it returns an object with '**value**' and '**done**' properties. 'done' is boolean, this will return true after reading all items in the collection.

Example Set and Iterator: 

```
var set = new Set(['a','b','c','d','e']);
// we can use set.keys(); or set.entries();
var iterator = set.values();
console.log(iterator.next());
```

Example Map and Iterator: 

```
var map = new Map([[1,'one'],[2,'two'],[3,'three']]);
// we can use set.keys(); or set.entries();
var iterator = set.values();
console.log(iterator.next());
```

## VII. Classes

Classes can be created using the class keyword in ES6. Classes can be included in the code either by declaring them or by using class expressions.

A class definition can include the following –

- **Constructors** – Responsible for allocating memory for the objects of the class.
- **Functions** – Functions represent actions an object can take. They are also at times referred to as methods.

These components put together are termed as the data members of the class.

**Note** – A class body can only contain methods, but not data properties.

Example: 

```
var Polygon = class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}
```

**Note** – Unlike variables and functions, classes cannot be hoisted.

### - Create an object

Example: 

```
Var obj = new Polygon(23,53);
```

### - Class Inheritance

ES6 supports the concept of **Inheritance**. Inheritance is the ability of a program to create new entities from an existing entity - here a class. Inheritance can be classified as –

- **Single** – Every class can at the most extend from one parent class.
- **Multiple** – A class can inherit from multiple classes. ES6 doesn't support multiple inheritance.
- **Multi-level** – Consider the following example.

Example: 'use strict'  

```
class Root {
    test() {
        console.log("call from parent class")
    }
}
```

### - The Static Keyword

The static keyword can be applied to functions in a class. Static members are referenced by the class name. **Note** – It is not mandatory to include a constructor definition. Every class by default has a constructor by default.

Example: 'use strict'  

```
class StaticMem {
    static disp() {
        console.log("Static Function called")
    }
}
StaticMem.disp() //invoke the static method
```

```
}
class Child extends Root {}
class Leaf extends Child {}

//indirectly inherits from Root by virtue of inheritance {}
```

```
var obj = new Leaf();
obj.test();
```

#### - Method Overriding

**Method Overriding** is a mechanism by which the child class redefines the superclass method.

Example: 'use strict';

```
class Person {
  print() {
    console.log("Person");
  }
}
class Student extends Person {
  print() {
    console.log("Student");
  }
}
var obj = new Student();
obj.print();
```

#### - Super keyword

ES6 enables a child class to invoke its parent class data member. This is achieved by using the **super** keyword. The super keyword is used to refer to the immediate parent of a class.

Example: 'use strict';

```
class Person {
  print() {
    console.log("Person");
  }
}
```

```

}
}
class Student extends Person {
  super.print();
  console.log("Student");
}
var obj = new Student();
obj.print();
```

#### - Getter and Setter

Example: class People {

```

  constructor(name) {
    this.name = name;
  }
  getName() {
    return this.name;
  }
  setName(name) {
    this.name = name;
  }
}
let person = new People("Jon Snow");
console.log(person.getName());
person.setName("Dany");
console.log(person.getName());
```

## VIII. Promise

**Promises** are a clean way to implement async programming in JavaScript (ES6 new feature). Prior to promises, Callbacks were used to implement async programming.

#### - Callback

A function may be passed as a parameter to another function. This mechanism is termed as a **Callback**.

Example: <script>

```
function notifyAll(fnSms, fnEmail) {
  console.log('starting notification process');
  fnSms();
  fnEmail();
}
notifyAll(function() {
  console.log("Sms send ..");
},
function() {
  console.log("email send ..");
});
console.log("End of script");
//executes last or blocked by other methods
</script>
```

#### - AsyncCallback

The setTimeout() method takes two parameters –

- A callback function.

- The number of seconds after which the method will be called.

Example: <script>

```
function notifyAll(fnSms, fnEmail) {
    setTimeout(function() {
        console.log('starting notification process');
        fnSms();
        fnEmail();
    }, 2000);
}
notifyAll(function() {
    console.log("Sms send ..");
},
function() {
    console.log("email send ..");
});
console.log("End of script"); //executes first or not blocked by
others
</script>
```

In multiple callback, ES6 comes to your rescue by introducing the concept of promises. Promises are "Continuation events" and they help you execute the multiple async operations together in a much cleaner code style.

Example: <script>

```
function getSum(n1, n2) {
    var isAnyNegative = function() {
        return n1 < 0 || n2 < 0;
    }
    var promise = new Promise(function(resolve,
reject) {
        if (isAnyNegative()) {
            reject(Error("Negatives not supported"));
        }
        resolve(n1 + n2);
    });
    return promise;
}
getSum(5, 6)
.then(function(result) {
    console.log(result);
    return getSum(10, 20);
    //this returns another Promise
},
```

```
function(error) {
    console.log(error);
})
.then(function(result) {
    console.log(result);
    return getSum(30, 40);
    //this returns another Promise
},
function(error) {
    console.log(error);
})
.then(function(result) {
    console.log(result);
},
function(error) {
    console.log(error);
});
console.log("End of script ");
</script>
```

## IX. Module

### - Exporting a Module

To make available certain parts of the module, use the export keyword.

Export a single value or element - Use export default

```
export default element_name
```

Export multi value or elements

```
export {element_name1, ...}
```



### - Import a Module

To be able to consume a module, use the import keyword. Following is the syntax for the same.

Import a single value or element

`import element_name from module_name`

Export multiple values or elements

`import {element_name1, ...} from`

`module_name`