This post contains list of important ES6 questions asked in the interviews.

**1. What is ES or ECMAScript or ES6?**

When JavaScript was developed, the name itself was chosen for marketing reasons since Java was getting popular around the time. But to standardize the language and its specification, it was submitted to ECMA international (European Computer Manufacturers Association), a body for standardization of information and communication technology. Eventually the language standardized in ECMA is called ECMAScript of ES. The first edition was released on June 1997.

The ES6 is the sixth edition of the language and was released on June 2015. It was initally known as ECMAScript 6 (ES6) and later renamed to ECMAScript 2015. This edition includes many new features like class, modules, iterators, for/of loop, arrow functions, typed arrays, promises, reflection.

In the next year on June 2016, ECMAScript 6 (ES6) was released and later renamed to ECMAScript 2015. This edition intended to continue the themes of language reform, includes two new features: exponentiation operator (\*\*) and Array.prototype.includes.

And then on June 2017, ECMAScript 2017 (ES2017), the eighth edition released which includes features for concurrency and atomics, syntactic integration with promises (async/await).

**2. What are some of the features of ES6?**

Some of the new features of ES6 are:

* Support for constants (also known as "immutable variables")
* Block-Scope support for both variables, constants, functions
* Arrow functions/li>
* Extended Parameter Handling
* Template Literals and Extended Literals
* Enhanced Regular Expression
* Destructuring Assignment
* Modules, Classes, Iterators, Generators
* Enhanced Object Properties
* Support for Map/Set & WeakMap/WeakSet
* Promises, Meta-Programming ,Internationalization and Localization

**3. How to use ES6?**

Since its release, many JavaScript environments like browsers and Node.js are actively working on implementing the language features. As of now, the Chrome and Safari has nearly 100% support for ES6 features. And also ES6 is backward compatible which means our old code is still valid in ES6. But in order to use it across all platforms, we need to transpile (transformation + compiling) the ES6 code to ES5. Babel is an awesome tool that let us do this job.

We can install babel globally through npm and can integrate it with our task runner or bundler like webpack to automatically transpiles our code into ES5 for production.

**4. What is let and const? And how it differs from var?**

Previously, when we declare any variable using **var**, it was function scoped. Meaning the variable can be accessed within the function. This leads to wrap the code in a function whenever we need to create a new scope.

But **let** and **const** uses block scoping. This means the variable declared using these keywords only exist within the innermost block that surronds them.

let a = 5;

{

let a = 3;

console.log('inner a:', a); *// inner a: 3*

}

console.log('outer a: ', a) *// outer a: 5;*

If we declare a **let** variable inside a block like if condition, for loop, it can be accessed within the block.

if(true) {

let i = 0;

console.log(i); *// prints 0;*

}

console.log(i); *// throws ReferenceError: i is not defined*

Also we can not re-declare the same variable with the same scope.

{

let a = 0;

console.log(a); *// 0*

let a = 1; *//TypeError: Identifier 'a' has already been declared*

}

Unlike **const**, **const** is immutable. It means the value must be given at the time of the declaration and it can not be re-assigned or changes. Although we can not change the value of the **const** but we can mutate them.

{

const a = 0;

a = 1; *// TypeError: Assignment to constant variable*

const b = [ 1, 2 ];

b.push(3); *// [1, 2, 3]*

b[3] = 4; *// [1, 2, 3, 4]*

}

It is always a good practice to use **let** and **const** over **var** and if we are not changing the value of the variable we should use **const**.

**5. What is Arrow function? What are all its uses? How it differs from normal function?**

Arrow functions are a short-hand notation for writing functions in ES6. The arrow function definition consists of a parameter list **( ... )**, followed by the **=>** marker and a function body. For single-argument functions, the parentheses may be omitted.

*// Classical Function Expression*

function add(a, b) {

return a + b;

};

*// Implementation with arrow function*

const add = (a, b) => a + b;

*// With single argument, no parentheses required*

const add5 = a => 5 + a;

If the arrow function is implemented with "concise body" (without **{}**), it does not need an explicit **return** statement. Note the omitted **{ }** after the **=>**.

Arrow functions behavior with **this** keyword varies from that of normal functions. Each function in JavaScript defines its own this context but arrow functions capture the **this** value of the nearest enclosing context.

function Timer() {

this.seconds = 0;

setInterval(() => {

setTimeout(() => {

this.seconds++; *// `this` properly refers to the Timer object*

}, 1000);

}, 1000);

}

There are four fundamental differences between arrow functions and **function** functions

* They close over **this**, and do not have their own versions.
* They can have a concise body (without **{ }**) rather than a verbose one (but they can have a verbose body as well).
* They cannot be used as constructors. E.g., you can't use **new** with an arrow function. Hence arrow functions do not have a **protoype** property on them.
* There is no generator syntax for arrow functions. E.g., there is no arrow equivalent to **function \*foo() { ... }**.

**6. What is Default Function Parameters? How can you handle the parameters of a function which is not passed to it?**

ES6 allows you to set default parameters in function definitions. **Default function parameters** allow formal parameters to be initialized with default values if no value or **undefined** is passed. Here is a simple illustration.

const increment = (number, add = 1) => number + 1;

increment(5); *// 6*

**7. What are Spread and Rest Operators in ES6?**

In ES6, **...** operator is referred to as spread or rest operator, depending on how and where it is used.

When it used with any iterable such as array expression, objects or string, it acts as to "spread" it into individual elements:

function sum(x, y, z) {

return x + y + z;

}

const numbers = [1, 2, 3];

console.log(sum(...numbers));

*// 6*

console.log(sum.apply(null, numbers));

*// 6 -> without ES6*

Spread syntax is also great when we need to create a new array or object from the existing one as part of it.

let parts = [ 3, 4 ];

let series = [ 1, 2, ...parts, 5 ];

*// [1, 2, 3, 4, 5]*

let defaults = { active: false };

let userData = { userName: 'foo', ...defaults };

*//userData = { userName: "foo", active: false }*

The **rest parameter** syntax allows us to represent an indefinite number of arguments as an array. This is also referred as **...**.

function sortRestArgs(...theArgs) {

var sortedArgs = theArgs.sort();

return sortedArgs;

}

console.log(sortRestArgs(5, 3, 7, 1)); *// 1, 3, 5, 7*

**8. What are all the new changes in Object literal notations in ES6?**

ES6 allows declaring object literals by providing shorthand syntax for initializing properties from variables and defining function methods. It also enables the ability to have computed property keys in an object literal definition.

var a = 'foo', b = 42, c = {};

*// shorthand for var o = { a: a, b: b, c: c }*

var o = {a, b, c};

*// shorthand method*

var o = {

add(a, b) { return a + b }

};

*// computed property names*

var prop = 'foo';

var o = {

[prop]: 'bar'

};

*// equals var o = { 'foo': 'bar' }*

**9. How do you use octal and binary numbers in ES6?**

ES6 has new support for octal and binary literals. Prependending a number with 0o or 0O would convert it into octal value. Have a look at the following code:

let oValue = 0o12;

*// 10*

let bValue = 0b10;

*// 2*

**10. How do you destructure an array or object in ES6? What is the use of array or object destructure**

Using the spread **...** syntax in ES6, we can destructure an array or object and store their respective values in separate variables. Destructuring helps in avoiding the need for temp variables when dealing with object and arrays.

function foo() {

return [1, 2, 3];

}

let arr = foo(); *// [1,2,3]*

let [a, b, c] = foo();

console.log(a, b, c); *// 1 2 3*

function bar() {

return {

x: 4,

y: 5,

z: 6

};

}

let { x: a, y: b, z: c } = bar();

console.log(a, b, c); *// 4 5 6*

**11. What is super keyword in ES6?**

The **super** keyword is used to access and call functions on an object's parent.

The **super.prop** and **super[expr]** expressions are valid in any method definition in both classes and object literals.

super([arguments]); *// calls the parent constructor.*

super.functionOnParent([arguments]);

Following is a simple example:

const parent = {

foo() {

console.log("Hello from the Parent");

}

}

const child = {

foo() {

super.foo();

console.log("Hello from the Child");

}

}

Object.setPrototypeOf(child, parent);

child.foo();

*// Hello from the Parent*

*// Hello from the Child*

**12. What is tempalte literals in ES6?**

Template literals are string literals allowing embedded expressions. We can use multi-line strings and string interpolation features with them.

ES6 introduces this easier way to add interpolations which are evaluated automatically.

* **`${ ... }`** is used for rendering the variables.
* **`** Backtick is used as delimiter.

let user = 'John Doe';

console.log(`Hi ${user}!`); *// Hi John Doe!*

Any newline characters inserted in the source are part of the template literal. Using normal strings, you would have to use the following syntax in order to get multi-line strings.

console.log('string text line 1

' +

'string text line 2');

*// "string text line 1*

*// string text line 2"*

To get the same effect using template literals, you can now write:

console.log(`string text line 1

string text line 2`);

*// "string text line 1*

*// string text line 2"*

A more advanced form of template literals are tagged templates. Tags allow you to parse template literals with a function. The first argument of a tag function contains an array of string values. The remaining arguments are related to the expressions.

var person = 'Mike';

var age = 28;

function myTag(strings, personExp, ageExp) {

var str0 = strings[0]; *// "That "*

var str1 = strings[1]; *// " is a "*

*// There is technically a string after*

*// the final expression (in our example),*

*// but it is empty (""), so disregard.*

*// var str2 = strings[2];*

var ageStr;

if (ageExp > 99){

ageStr = 'centenarian';

} else {

ageStr = 'youngster';

}

*// We can even return a string built using a template literal*

return `${str0}${personExp}${str1}${ageStr}`;

}

var output = myTag`That ${ person } is a ${ age }`;

console.log(output);

*// That Mike is a youngster*

**13. Explain for...in and for...of loops and it differences?**

The **for...of** loop iterating over iterable objects (like string, array, NodeList) for values.

let iterable = [10, 20, 30];

for (const value of iterable) {

console.log(value);

}

*// 10*

*// 20*

*// 30*

The **for...in** statement iterates over all non-Symbol, enumerable properties of an object.

var object1 = {a: 1, b: 2, c: 3};

for (let key in object1) {

console.log(object1[key]);

}

*// 1*

*// 2*

*// 3*

**14. What is Map in ES6?**

The **Map** object holds key-value pairs. Any value (both objects and primitive values) may be used as either a key or a value. A **Map** object iterates its elements in insertion order. **Objects** are similar to **Maps** in that both let you set keys to values, retrieve those values, delete keys, and detect whether something is stored at a key. However, there are important differences between them.

* The keys of an **Object** are **Strings** and **Symbols**, whereas they can be any value for a **Map**, including functions, objects, and any primitive.
* The keys in **Map** are ordered while keys added to object are not. Thus, when iterating over it, a Map object returns keys in order of insertion.
* You can get the size of a **Map** easily with the size property, while the number of properties in an Object must be determined manually.
* A **Map** is an iterable and can thus be directly iterated, whereas iterating over an Object requires obtaining its keys in some fashion and iterating over them.

var myMap = new Map();

var keyString = 'a string',

keyObj = {},

keyFunc = function() {};

*// setting the values*

myMap.set(keyString, "value associated with 'a string'");

myMap.set(keyObj, 'value associated with keyObj');

myMap.set(keyFunc, 'value associated with keyFunc');

myMap.size; *// 3*

*// getting the values*

myMap.get(keyString); *// "value associated with 'a string'"*

myMap.get(keyObj); *// "value associated with keyObj"*

myMap.get(keyFunc); *// "value associated with keyFunc"*

myMap.get('a string'); *// "value associated with 'a string'"*

*// because keyString === 'a string'*

myMap.get({}); *// undefined, because keyObj !== {}*

myMap.get(function() {}) *// undefined, because keyFunc !== function () {}*

**15. What is WeakMap in ES6?**

The **WeakMap** is same as **Map** where it is a collection of key/value pairs. But in **WeakMap**, the **keys must be objects** and the values can be arbitrary values. The object references in the keys are held weakly, meaning that they are a target of garbage collection (GC) if there is no other reference to the object anymore. The **WeakMap** API is the same as the **Map** API.

However, One difference to Map objects is that WeakMap keys are not enumerable. And there are no methods giving us a list of keys. If they were, the list would depend on the state of garbage collection, introducing non-determinism. If we want to have a list of keys, we should use a **Map**.

var wm1 = new WeakMap();

var o1 = {};

wm1.set(o1, 37);

wm1.get(o1); *// 37*

wm1.has(o1); *// true*

wm3.set(o1, 40);

wm3.get(o1); *// 40*

wm1.has(o1); *// true*

wm1.delete(o1);

wm1.has(o1); *// false*

**16. What is Set?**

**Set** objects are collections of unique values. Duplicate values are ignored, as the collection must have all unique values. The values can be primitive types or object references.

var mySet = new Set();

mySet.add(1); *// Set [ 1 ]*

mySet.add(5); *// Set [ 1, 5 ]*

mySet.add(5); *// Set [ 1, 5 ] -- ignored*

mySet.add('some text'); *// Set [ 1, 5, 'some text' ]*

var o = {a: 1, b: 2};

mySet.add(o);

mySet.size; *// 4*

console.log(mySet); *// Set [ 1, 5, 'some text', Object {a: 1, b: 2} ]*

Also, **NaN** and **undefined** can also be stored in a **Set**. **NaN** is considered the same as **NaN** (even though **NaN !== NaN**).

**17. What is WeakSet?**

The **WeakSet** object lets you store weakly held objects in a collection.

**WeakSet** objects are collections of objects. An object in the **WeakSet** may occur only once; it is unique in the **WeakSet**'s collection. The main differences to the **Set** object are:

* Unlike **Set**, **WeakSet**s are collections of **objects only** and not of arbitrary values of any type.
* The **WeakSet** is weak: References to objects in the collection are held weakly. If there is no other reference to an object stored in the **WeakSet**, they can be garbage collected. That also means that there is no list of current objects stored in the collection. WeakSets are not enumerable.

var ws = new WeakSet();

var window = {};

var foo = {};

ws.add(window);

ws.add(obj);

ws.has(window); *// true*

ws.has(foo); *// false, foo has not been added to the set*

ws.delete(window); *// removes window from the set*

ws.has(window); *// false, window has been removed*

**18. Explain Classes in ES6?**

ES6 introduces new **Class** syntax. However, ES6 class is not a new object-oriented inheritance model. They just serve as a syntactical sugar over JavaScript's existing prototype-based inheritance.

One way to look at a class in ES6 is just a new syntax to work with prototypes and contructor functions that we'd use in ES5. Functions defined using the **static** keyword implement static/class functions on the class. The following is an example of class declaration:

class Polygon {

constructor(height, width) {

this.area = height \* width;

}

}

console.log(new Polygon(4,3).area);

*// expected output: 12*

The **Class** body of a class declaration is executed in strict mode. The constructor property is optional. We can also use **extend** to allow child class to inherit from parent class in ES6. It is important to note that the derived constructor must call **super()**.

class Car {

constructor() {

console.log("Creating a new car");

}

}

class Porsche extends Car {

constructor() {

super();

console.log("Creating Porsche");

}

}

let c = new Porsche();

*// Creating a new car*

*// Creating Porsche*

We can not re-declare the class again and trying to do so will throw SyntaxError. **Class** declarations are not hoisted. You first need to declare your class and then access it, otherwise ReferenceError will be thrown. Also There is no need to use function keyword when defining functions inside a class definition.

**19. What is Class expression?**

The **Class** expression is one way to define a class in ES6. Similar to function expressions, class expressions can be named or unnamed. If named, the name of the class is local to the class body only. JavaScript classes use prototype-based inheritance.

var Rectangle = class {

constructor(height, width) {

this.height = height;

this.width = width;

}

area() {

return this.height \* this.width;

}

}

console.log(new Rectangle(5,8).area());

*// expected output: 40*

A class expression has a similar syntax to a class statement (declaration). However, with class expressions, you are able to omit the class name ("binding identifier"), which you can't with class statements. Additionally, class expressions allow you to redefine/re-declare classes and don't throw any type errors like class declaration. The constructor property is optional. And, typeof the classes generated using this keyword will always be "function".

**20. What is Symbol in ES6?**

A **Symbol** is a unique and immutable data type introduced in ES6. The purpose of a symbol is to generate a unique identifier but you can never get any access to that identifier. Symbol is a primitive value and can have an optional description, but for debugging purposes only.

Here’s how you create a symbol:

const sym = Symbol("some optional description");

console.log(typeof sym); *// symbol*

Note that you cannot use **new** with **Symbol()**. If a symbol is used as a property/key of an object, it’s stored in a special way that the property will not show up in a normal enumeration of the object’s properties.

const o = {

val: 10,

[Symbol("random")]: "I'm a symbol",

};

console.log(Object.getOwnPropertyNames(o)); *// val*

To retrieve an object’s symbol properties, use **Object.getOwnPropertySymbols(o)**.

**21. What is iterators in ES6?**

An iterator accesses the items from a collection one at a time, while keeping track of its current position within that sequence. It provides a **next()** method which returns the next item in the sequence. This method returns an object with two properties: **done** and **value**.

ES6 has **Symbol.iterator** which specifies the default iterator for an object. Whenever an object needs to be iterated (such as at the beginning of a **for..of** loop), its **@@iterator** method is called with no arguments, and the returned iterator is used to obtain the values to be iterated.

const arr = [11,12,13];

const itr = arr[Symbol.iterator]();

itr.next(); *// { value: 11, done: false }*

itr.next(); *// { value: 12, done: false }*

itr.next(); *// { value: 13, done: false }*

itr.next(); *// { value: undefined, done: true }*

Some built-in types have a default iteration behavior, while other types (such as Object) do not. The built-in types with a **@@iterator** method are:

* **Array.prototype[@@iterator]()**
* **TypedArray.prototype[@@iterator]()**
* **String.prototype[@@iterator]()**
* **Map.prototype[@@iterator]()**
* **Set.prototype[@@iterator]()**

We can make our own iterables for objects like below:

var myIterable = {}

myIterable[Symbol.iterator] = function\* () {

yield 1;

yield 2;

yield 3;

};

[...myIterable] *// [1, 2, 3]*

**22. What is Generator function?**

Generator functions are a new feature in ES6 that allow a function to generate many values over time by returning an object which can be iterated over to pull values from the function one value at a time.

A generator function returns an **iterable object** when it's called. It is written using the new **\*** syntax as well as the new yield keyword introduced in ES6.

function \*infiniteNumbers() {

let n = 1;

while (true) {

yield n++;

}

}

const numbers = infiniteNumbers(); *// returns an iterable object*

numbers.next(); *// { value: 1, done: false }*

numbers.next(); *// { value: 2, done: false }*

numbers.next(); *// { value: 3, done: false }*

Each time yield is called, the yielded value becomes the next value in the sequence. Also, note that generators compute their yielded values on demand, which allows them to efficiently represent sequences that are expensive to compute, or even infinite sequences.

**23. Explain Promises in ES6?**

ES6 has native support for promises. A promise is an object that is waiting for an asynchronous operation to complete, and when that operation completes, the promise is either fulfilled(resolved) or rejected.

var promise1 = new Promise(function(resolve, reject) {

setTimeout(resolve, 100, 'foo');

});

console.log(promise1);

*// expected output: [object Promise]*

A Promise is a proxy for a value not necessarily known when the promise is created. It allows us to associate handlers with an asynchronous action's eventual success value or failure reason. This lets asynchronous methods return values like synchronous methods: instead of immediately returning the final value, the asynchronous method returns a promise to supply the value at some point in the future. A Promise is in one of these states:

* pending: initial state, neither fulfilled nor rejected.
* fulfilled: meaning that the operation completed successfully.
* rejected: meaning that the operation failed.

A pending promise can either be fulfilled with a value, or rejected with a reason (error). When either of these options happens, the associated handlers queued up by a promise's then method are called.

const p = new Promise((resolve, reject) => {

if (someCondition) {

resolve(promiseResolveValue); *// fulfilled successfully*

} else {

reject(promiseRejectvalue); *// error, rejected*

}

});

p.then((val) => console.log("Promise Resolved", val),

(err) => console.log("Promise Rejected", err));

When returning a promise, the resolved value of the promise will get passed to the next callback to effectively chain them together. This is a simple technique to avoid "callback hell".

const hello = new Promise((resolve, reject) => { resolve("Hello") });

hello.then((str) => `${str} World`)

.then((str) => `${str}!`)

.then((str) => console.log(str)) *// Hello World!*

**24. What is Proxy and Reflect in ES6?**

ES6 introduced **Proxy** objects allow you to intercept certain operations and to implement custom behaviors. For example getting a property on an object:

var handler = {

get: function(target, name) {

return name in target ? target[name] : 42;

}

};

var p = new Proxy({}, handler);

p.a = 1;

console.log(p.a, p.b); *// 1, 42*

The **Proxy** object defines a target (an empty object here) and a handler object in which a **get** trap is implemented. Here, an object that is proxied will not return **undefined** when getting undefined properties, but will instead return the number **42**.

**Reflect** is a built-in object that provides methods for interceptable JavaScript operations. The methods are the same as those of the proxy handlers. **Reflect** is not a function object.

**Reflect** helps with forwarding default operations from the handler to the target. With **Reflect.has()** for example, you get the in operator as a function:

Reflect.has(Object, 'assign'); *// true*

**Reflect** can also be used instead of **Function.prototype.apply()** method to call a function with a given **this** value and **arguments** provided as an array. With **Reflect** this becomes less verbose and easier to understand:

*// instead of*

Function.prototype.apply.call(Math.floor, undefined, [1.75]);

Reflect.apply(Math.floor, undefined, [1.75]); *// 1*

**25. Explain async and await?**

**async** functions were introduced in ES8 (ECMAScript2017) and serve as a new feature for cleaner handling of Promises responses, errors and more.

The **async** function declaration defines an asynchronous function, which returns an AsyncFunction object. An asynchronous function is a function which operates asynchronously via the event loop, using an implicit **Promise** to return its result. But the syntax and structure of your code using async functions is much more like using standard synchronous functions.

function resolveAfter2Seconds() {

return new Promise(resolve => {

setTimeout(() => {

resolve('resolved');

}, 2000);

});

}

async function asyncCall() {

console.log('calling');

var result = await resolveAfter2Seconds();

console.log(result);

*// expected output: "resolved"*

}

asyncCall();

An async function can contain an **await** expression that pauses the execution of the **async** function and waits for the passed Promise's resolution, and then resumes the **async** function's execution and returns the resolved value.

Remember, the **await** keyword is only valid inside **async** functions. If we use it outside of an **async** function's body, we will get a **SyntaxError**.