# LEARN JAVASCRIPT(ES6) IN ONEDAY

Basics Of JavaScript 6

Abstract

No prior knowledge of ES5 or JavaScript5 is required.

Haramohan Sahu

hara.sahu@gmail.com

## Contents

## New ES6 Features:

### let

In ES5, when you declare a variable using the var keyword, the scope of the variable is global if you declare it outside of a function or local in case you declare it inside a function.

ES6 provides a new way of declaring a variable by using the let keyword. The let keyword is similar to the var keyword, except that the variables it declares are block-scoped:

```
let x = 10;
if (x == 10) {
    let x = 20;
    console.log(x); // 20:  reference x inside the block
}
console.log(x); // 10: reference at the beginning of the script
```

### JavaScript **let** and global object:

When you declare a global variable using the var keyword, you are adding that variable to the property list of the global object. In the case of the web browser, the global object is the window.

See the following example:

```
var a = 10;
console.log(window.a); // 10
```

However, when you use the let keyword to declare a variable, that variable is not attached to the global object as a property.

Here is an example:

```
let b = 20;
console.log(window.b); // undefined
```

### JavaScript let anallback function in a for loop

See the following example.

```
for (var i = 0; i < 5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
```

}
Here the output is 5 gets printed , 5 times.

- setTimeout() is a method of the window object.
- setTimeout() sets a timer and executes a callback function when the timer expires.

reason is that after five iterations, the value of the i variable is 5. And the five instances of the callback function passed to the setTimeOut() function references the same variable i with the final value 5.

you fix this issue by creating another scope so that each instance of the callback function references a new variable. And to create a new scope, you need to create a function.

```
for (var i = 0; i < 5; i++) {
    (function (j) {
        setTimeout(function () {
            console.log(j);
        }, 1000);
    })(i);
}
```

In ES6, the let keyword declares a new variable in each loop iteration, therefore, you just need to replace the var keyword by the let keyword to fix the issue.

```
for (let i = 0; i < 5; i++) {
    setTimeout(function () {
        console.log(i);
    }, 1000);
}
```

Same Using arrow function:

```
for (let i = 0; i < 5; i++) {
    setTimeout(() => console.log(i), 1000);
}
```

## Redeclaration using let

The var keyword allows you to re declare a variable without any issue:

```
var counter = 0;
var counter;
console.log(counter); // 0
```

However, re declaring a variable using the let keyword will result in an error:

```
let counter = 0;
let counter;
console.log(counter);
```

Here is the error message:

```
Uncaught SyntaxError: Identifier 'counter' has already been declared
```

## JavaScript **let** variables and hoisting:

```
console.log(counter); // Uncaught ReferenceError: Cannot access 'counter' before
                        initialization
let counter = 10;
```

OUTPUT:

```
Uncaught ReferenceError: Cannot access 'counter' before initialization
```

In the function, accessing the counter variable before declaring it causes a ReferenceError. In fact, the JavaScript engine will hoist the variables declared using the let keyword to the top of the block. However, it does not initialize the variables. Therefore, when you reference uninitialized variables, you get a ReferenceError.

- Variables are declared using the let keyword are:
  - block-scoped
  - not initialized to any value.
  - not attached to the global object.
- Redeclaring a variable using the let keyword will cause an error.
- A temporal dead zone of a variable declared using the let keyword starts from the block until the initialization is evaluated.

Last but not least, from ES6, it is recommended that you should adapt let keyword and stop using the var keyword when you declare a variable.

## Differences Between var and let:

- Variable scopes:
  - Variable declared with var will have global scope when it is declared outside of a function.
  - Variable declared with let will have block scope.

```
for (var i = 0; i < 5; i++) {
   console.log(`Inside the loop: ${i}`);
}

console.log(`Outside the loop: ${i}`);
```

Here I declared with var, will accessed outside of the  loop. But not the same if it is declared with LET.

- Creating global properties:
  - The global var variables are added to the global object as properties. The global object is window on the web browser and global on Node.js:
- Re-declaration
  - The var keyword allows you to redeclare a variable without any issue:

Example:
var counter = 10;
var counter;
console.log(counter); // 10

However, if you redeclare a variable with the let keyword, you will get an error:
let counter = 10;
let counter; // error

## The **var** variables

- In the creation phase, the var variables are assigned storage spaces and immediately initialized to undefined.
- In the execution phase, the var variables are assigned the values specified by the assignments if there are ones. If there aren't, the values of the variables remain undefined.

## The **let** variables

- In the creation phase, thelet variables are assigned storage spaces but are not initialized. Referencing uninitialized variables will cause a ReferenceError.
- The let variables have the same execution phase as the var variables.

## Introduction to the JavaScript const keyword

The const keyword creates a read-only reference to a value. The const keyword works like the let keyword. But the const keyword creates block-scoped variables whose values can't be **reassigned**.

**JavaScript const and Objects**

The const keyword ensures that the variable it creates is read-only. However, it doesn't mean that the actual value to which the const variable reference is immutable.

For example:

```
const person = { age: 20 };
person.age = 30; // OK
console.log(person.age); // 30
```

However, you cannot reassign a different value to the person constant like this:

```
person = {age: 40}; // TypeError
```

If you want the value of the person object to be immutable, you have to freeze it by using the Object.freeze() method:

```
const person = Object.freeze({age: 20});
person.age = 30; // TypeError
```

Note that Object.freeze() is shallow, meaning that it can freeze the properties of the object, not the objects referenced by the properties. For example, the company object is constant and frozen.

```
const company = Object.freeze({
    name: 'ABC corp',
    address: {
        street: 'North 1st street',
        city: 'San Jose',
        state: 'CA',
        zipcode: 95134
    }
});
```

But the company.address object is not immutable, you can add a new property to the company.address object as follows:

```
company.address.country = 'USA'; // OK
```

## JavaScript const and Arrays

Consider the following example:

```
const colors = ['red'];
colors.push('green');
console.log(colors); // ["red", "green"]

colors.pop();
colors.pop();
console.log(colors); // []

colors = []; // TypeError
```

## Usages of JavaScript Spread Operator:

ES6 provides a new operator called spread operator that consists of three dots (...). The spread operator allows you to spread out elements of an iterable object such as an array, a  map, or a set.

For example:

```
const odd = [1,3,5];
const combined = [2,4,6, ...odd];
console.log(combined); // [ 2, 4, 6, 1, 3, 5 ]
```

In this example, the three dots (...) located in front of the odd array is the spread operator. The spread operator unpacks the elements of the odd array.

**Note that ES6 also has the three dots (...) which is a rest parameter that collects all remaining arguments of a function into an array.**
**Note that ES2018 expands the spread operator to objects. It is known as the object spread.**

So the three dots ( ...) represent both the spread operator and the rest parameter.

## Usages of JavaScript Spread Rest Parameters:

The rest parameter allows you to represent an indefinite number of arguments as an array.

**Example:**

```
function fn(a,b,...args) {
  //...
}
```

The last parameter  ( args) is prefixed with the three-dots ( ...) is called the rest parameter ( ...args) All the arguments that you pass in the function will map to the parameter list. In the syntax above, the first argument maps to a, the second one maps to b, and the third, the fourth, etc., will be stored in the rest parameter args as an array.

**For example:**

```
fn(1,2,3,'A','B','C');
```

**Notice** that the rest parameters must be at the end of the argument list

Suppose, we have a function which takes rest parameters, but in the function logic, we need to calculate sum of the numbers. There is a high chance that caller of the function may pass values other then number which is a need to for the function.

Let’s have an example.

```
Function sumArg(…args)
```

```
{
let total = 0;
    for (const a of args) {
        total += a;
    }
    return total;
}
```

The above logic will not work, if user had called the function with sumArg(11,22,’haramohan sahu’,55);
So we need to write in a different way like below:

```
function sumArg (...args) {
    return args.filter(e => typeof e === 'number')
        .reduce((prev, curr)=> prev + curr);
}
alert(sumArg (1,2,3,'haramohan',4,'sahu',5));
```

Output will be 15

**JavaScript rest parameters and arrow function**

```
const combine = (...args) => {
    return args.reduce((prev, curr) => prev + ' ' + curr);
};
let message = combine('JavaScript', 'Rest', 'Parameters'); // =>
console.log(message); // JavaScript Rest Parameters
```

## The difference between Rest and spread operator in JavaScript

So the three dots ( ...) represent both the spread operator and the rest parameter. Here are the main differences:

- The spread operator unpacks elements.
- The rest parameter packs elements into an array.

The rest parameters must be the last arguments of a function. However, the spread operator can be anywhere:

**Rest parameters:**

```
function sum(a,b,...remaining){
     console.log(a);
     console.log(b);
     console.log(remaining);
}
```

Rest

```
sum(1,2,3,4,5,6,7);

> 1
> 2
> Array [3, 4, 5, 6, 7]
```

**Spread operator**

- Splitting the strings

```
let name = "JavaScript";
let arrayOfStrings = [...name];
console.log(arrayOfStrings);

 // Ouptut -> ["J", "a", "v", "a", "S", "c", "r", "i", "p", "t"]
```

- Merging arrays

```
const group1 = [1,2,3];
const group2 = [4,5,6];
const allGroups = [...group1,...group2];
console.log(allGroups)
//output -> [1, 2, 3, 4, 5, 6]
```

- Copying Arrays:

```
let scores = [80, 70, 90];
let copiedScores = [...scores];
console.log(copiedScores); // [80, 70, 90]
```

- Concatenating arrays

Also, you can use the spread operator to concatenate two or more arrays:

```
let numbers = [1, 2];
let moreNumbers = [3, 4];
let allNumbers = [...numbers, ...moreNumbers];
console.log(allNumbers); // [1, 2, 3, 4]
```

- Merging Objects

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

const merge  = {...obj1, ...obj2};
console.log(merge); // {a:1, b:2, c:3, d:4}
```

We can use the spread operator to pass an array of numbers as a individual function arguments.

```
function sum(a,b,c){
   return a+b+c;
}

const nums = [1,2,3];
//function calling
sum(...nums) // 6
```

## Object Literal Syntax Extensions in ES6

The object literal is one of the most popular patterns for creating objects in JavaScript because of its simplicity. ES6 makes the object literal more succinct and powerful by extending the syntax in some ways.

Object literals is nothing but collection of name-value pair:

```
function createMachine(name, status) {
   return {
      name: name,
      status: status
   };
}
```

In ES6, if the poperty name and local variable is same, then you can remove the duplicate & can be written below:

```
function createMachine(name, status) {
   return {
      name,
      status
   };
}
```

This way also we can create object literals:

1)
```
let name = 'Computer',
    status = 'On';

let machine = {
   name,
   status
};
```
2)
```
let name = 'machine name';
let machine = {
    [name]: 'server',
    'machine hours': 10000
};

console.log(machine[name]); // server
console.log(machine['machine hours']); // 10000
```

## JavaScript for...of Loop in ES6

ES6 introduced a new construct for...of that creates a loop iterating over iterable objects that include:

1. Built-in Array, String, Map, Set, …
2. Array-like objects such as arguments or NodeList
3. User-defined objects that implement the iterator protocol.

The following illustrates the syntax of the for...of:
```
for (variable of iterable) {
   // statements
}
```

## for of loop examples

1. **Iterating over arrays**

The following example shows how to use the for...of to iterate over elements of an array.

```
let scores = [80, 90, 70];
for (let score of scores) {
    score = score + 5;
    console.log(score);
}
```

To access the index of the array elements inside the loop, you can use the for...loop statement with the entries() method of the array.
The array.entries() method returns a pair of [index, element] in each iteration. For example:

```
let colors = ['Red', 'Green', 'Blue'];

for (const [index, color] of colors.entries()) {
   console.log(`${color} is at index ${index}`);
}
```

2. **In-place object destructuring with for…of**

```
const ratings = [
   {user: 'John',score: 3},
   {user: 'Jane',score: 4},
   {user: 'David',score: 5},
   {user: 'Peter',score: 2},
];

let sum = 0;
for (const {score} of ratings) {
   sum += score;
}

console.log(`Total scores: ${sum}`); // 14
```

**How it works:**

- The ratings is an array of objects. Each object has two properties user and score.
- The for...of iterate over the ratings array and calculate the total scores of all objects.
- The expression const {score} of ratings uses object destructing to assign the score property of the current iterated element to the score variable.

3. **Iterating over strings**

The following example uses the for...of loop to iterate over characters of a string.

```
let str = 'abc';
for (let c of str) {
   console.log(c);
}
```

4. **Iterating over Map objects**

The following example illustrates how to use the for...of statement to iterate over a Map object.

```
let colors = new Map();
colors.set('red', '#ff0000');
colors.set('green', '#00ff00');
colors.set('blue', '#0000ff');

for (let color of colors) {
   console.log(color);
}
```

5. for...of **vs.** for...in

- The for...in iterates over all enumerable properties of an object. It doesn't iterate over a collection such as Array, Map or Set.
- Unlike the for...in loop, the for...of iterates a collection, rather than an object.
- In fact, the for...of iterates over elements of any collection that has the [Symbol.iterator] property.

## JavaScript Rest Parameters:

ES6 provides a new kind of parameter so-called rest parameter that has a prefix of three dots (...). The rest parameter allows you to represent an indefinite number of arguments as an array. See the following syntax:

```
function fn(a, b, ...args) {
   //...
}
```

The last parameter ( args) is prefixed with the three-dots ( ...) is called the rest parameter ( ...args) All the arguments that you pass in the function will map to the parameter list. In the syntax above, the first argument maps to a, the second one maps to b, and the third, the fourth, etc., will be stored in the rest parameter args as an array.

For example:

```
fn(1,2,3,'A','B','C');
```

The args array stores the following values:

```
[3,'A','B','C']
```

Notice that the rest parameters must be at the end of the argument list. The following code causes an error:

```
function foo(a,...rest, b) {
 // error
};
```

**JavaScript rest parameter in a dynamic function**

JavaScript allows you to create dynamic functions through the Function constructor. And it is possible to use the rest parameter in a dynamic function.

Here is an example:

```
var showNumbers = new Function('...numbers', 'console.log(numbers)');
showNumbers(1, 2, 3);
```

**Output:**

```
[ 1, 2, 3 ]
```

## The arguments object

The value of the arguments object inside the function is the number of actual arguments that you pass into. See this example:

```
function add(x, y = 1, z = 2) {
    console.log(arguments.length );
    return x + y + z;
}

add (10); // 1
add (10, 20); // 2
add (10, 20, 30); // 3
```

## Destructuring Assignment:

ES6 destructuring assignment that allows you to destructure an array into individual variables.

It also allows us to destructure properties of an object or elements of an array into individual variables.

1)

Let discuss with an example:

```
function getScores() {
   return [70, 80, 90];
}
let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

2)

```
function getScores() {
   return [70, 80];
}

let [x, y, z] = getScores();

console.log(x); // 70
console.log(y); // 80
console.log(z); // undefined
```

3)

```
function getScores() {
   return [70, 80, 90, 100];
}
let [x, y, z] = getScores();
console.log(x); // 70
console.log(y); // 80
console.log(z); // 90
```

## Object Destructuring:

Object destructuring that assigns properties of an object to individual variables.
Suppose you have a person object with two properties: **firstName** and **lastName**.

```
let person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

ES6 introduces the object destructuring syntax that provides an alternative way to assign properties of an object to variables:

1)

```
let {
   firstName: fname,
   lastName: lname
} = person;
```

2)

```
let person = {
    firstName: 'John',
    lastName: 'Doe',
    middleName: "Mohan",
    currentAge: 28
};

let {
    firstName,
    lastName,
    middleName = '',
    currentAge
} = person;

console.log(middleName,firstName,lastName); // ''
console.log(currentAge); // 28
```

3)

```
let employee = {
    id: 1001,
    name: {
        firstName: 'John',
        lastName: 'Doe'
    }
};

let {
    name: {
        firstName,
        lastName
    },
    name
} = employee;

console.log(firstName); // John
console.log(lastName); // Doe
console.log(name); // { firstName: 'John', lastName: 'Doe' }
```

## From IIFEs to blocks

In ES5, you had to use a pattern called IIFE (Immediately-Invoked Function Expression) if you wanted to restrict the scope of a variable tmp to a block:

```
(function () {  // open IIFE
    var tmp = ···;
    ···
}());  // close IIFE

console.log(tmp); // ReferenceError
```

In ECMAScript 6, you can simply use a block and a let declaration (or a const declaration):

```
{  // open block
    let tmp = ···;
    ···
}  // close block


console.log(tmp); // ReferenceError
```

## ES6 template literals can span multiple lines:

```
const HTML5_SKELETON = `
    <!doctype html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
    </body>
    </html>`;
```

## ES6 – spread operator:

```
const arr1 = ['a', 'b'];
const arr2 = ['c', 'd'];
```

```
arr1.push(...arr2);
    // arr1 is now ['a', 'b', 'c', 'd']
```

## Base classes

In ES5, you implement constructor functions directly:

```
function Person(name) {
    this.name = name;
}
Person.prototype.describe = function () {
    return 'Person called '+this.name;
};
```

In ES6, classes provide slightly more convenient syntax for constructor functions:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    describe() {
        return 'Person called '+this.name;
    }
}
```

Note the compact syntax for method definitions – no keyword function needed. Also note that there are no commas between the parts of a class.

## Derived classes

Sub classing is complicated in ES5, especially referring to super-constructors and super-properties. This is the canonical way of creating a sub-constructor Employee of Person:

```
function Employee(name, title) {
    Person.call(this, name); // super(name)
    this.title = title;
}

Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
Employee.prototype.describe = function () {
    return Person.prototype.describe.call(this) // super.describe()
           + ' (' + this.title + ')';
};
```

ES6 has built-in support for sub classing, via the extends clause:

```
class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    describe() {
        return super.describe() + ' (' + this.title + ')';
    }
}
```

## Class declarations are not hoisted

Function declarations are *hoisted*: When entering a scope, the functions that are declared in it are immediately available – independently of where the declarations happen. That means that you can call a function that is declared later:

```
foo(); // works, because `foo` is hoisted

function foo() {}
```

In contrast, class declarations are not hoisted. Therefore, a class only exists after execution reached its definition and it was evaluated. Accessing it beforehand leads to a ReferenceError:

```
new Foo(); // ReferenceError

class Foo {}
```

The reason for this limitation is that classes can have an extends clause whose value is an arbitrary expression. That expression must be evaluated in the proper “location”, its evaluation can’t be hoisted.

Not having hoisting is less limiting than you may think. For example, a function that comes before a class declaration can still refer to that class, but you have to wait until the class declaration has been evaluated before you can call the function.

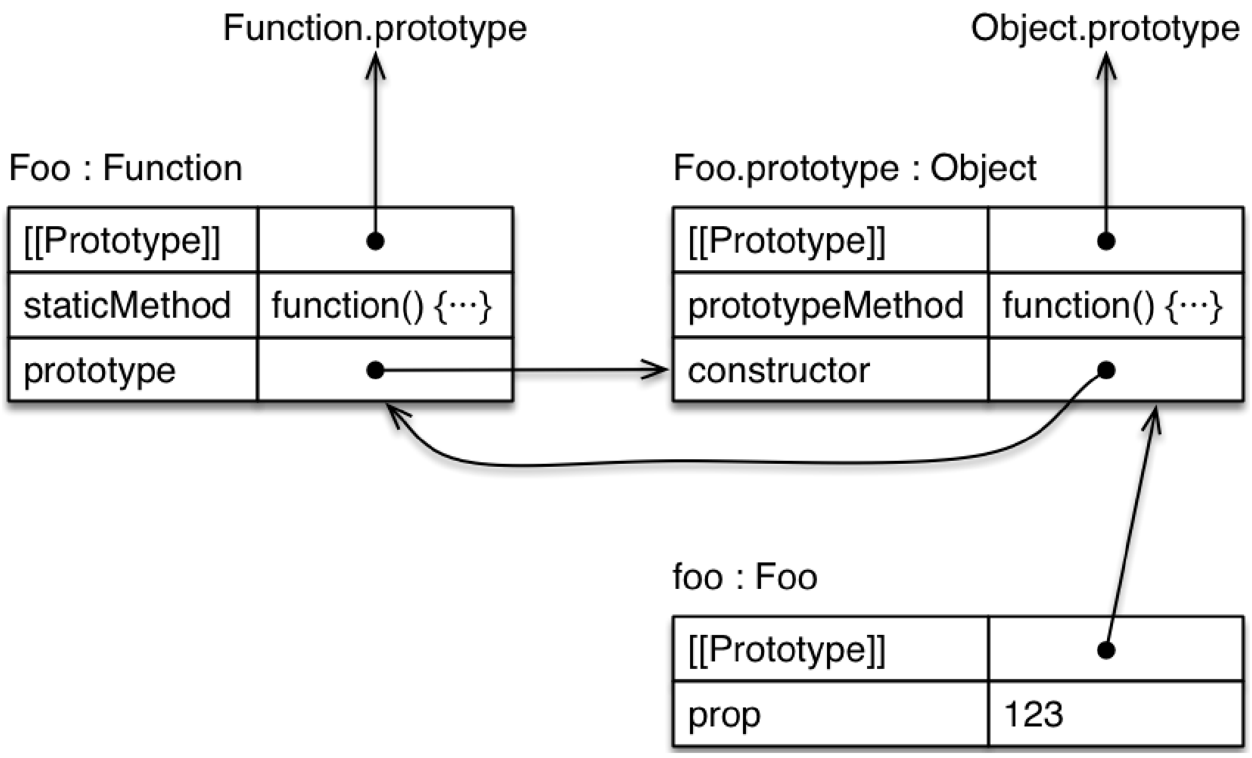
```
function functionThatUsesBar() {
    new Bar();
}

functionThatUsesBar(); // ReferenceError
class Bar {}
functionThatUsesBar(); // OK
```

## constructor, static methods, prototype methods

Let’s examine three kinds of methods that you often find in class definitions.

```
class Foo {
    constructor(prop) {
        this.prop = prop;
    }
    static staticMethod() {
        return 'classy';
    }
    prototypeMethod() {
        return 'prototypical';
    }
}
const foo = new Foo(123);
```

The object diagram for this class declaration looks as follows. Tip for understanding it: [[Prototype]] is an inheritance relationship between objects, while prototype is a normal property whose value is an object. The property prototype is only special w.r.t. the new operator using its value as the prototype for instances it creates.



**First, the pseudo-method constructor.** This method is special, as it defines the function that represents the class:

```
> Foo === Foo.prototype.constructor
true
> typeof Foo
'function'
```

It is sometimes called a class constructor. It has features that normal constructor functions don’t have (mainly the ability to constructor-call its superconstructor via super(), which is explained later).

**Second, static methods.** *Static properties* (or *class properties*) are properties of Foo itself. If you prefix a method definition with static, you create a class method:

```
> typeof Foo.staticMethod
'function'
> Foo.staticMethod()
'classy'
```

**Third, prototype methods.** The prototype properties of Foo are the properties of Foo.prototype. They are usually methods and inherited by instances of Foo.

```
> typeof Foo.prototype.prototypeMethod
'function'
> foo.prototypeMethod()
'prototypical'
```

## Generator methods

If you prefix a method definition with an asterisk (*), it becomes a *generator method*. Among other things, a generator is useful for defining the method whose key is Symbol.iterator. The following code demonstrates how that works.

```
class IterableArguments {
    constructor(...args) {
        this.args = args;
    }
    * [Symbol.iterator]() {
        for (const arg of this.args) {
            yield arg;
        }
    }
}

for (const x of new IterableArguments('hello', 'world')) {
    console.log(x);
}

// Output:
// hello
// world
```

The following table describes the attributes of properties related to a given class Foo:

| | **writable** | **enumerable** | **configurable** |
|---|---|---|---|
| Static properties Foo.* | true | false | true |
| Foo.prototype | false | false | false |
| Foo.prototype.constructor | false | false | true |
| Prototype properties Foo.prototype.* | true | false | true |

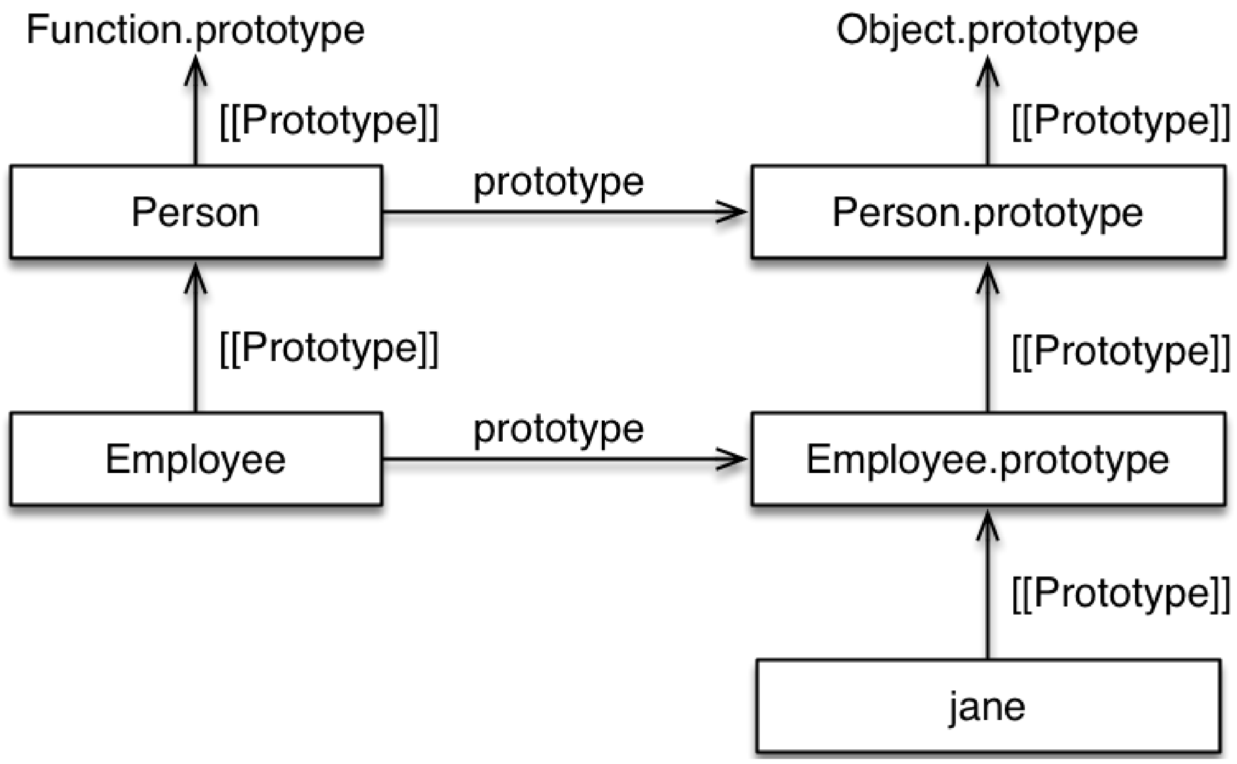
## The inner names of classes

Interestingly, ES6 classes also have lexical inner names that you can use in methods (constructor methods and regular methods):

```
class C {
    constructor() {
        // Use inner name C to refer to class
        console.log(`constructor: ${C.prop}`);
    }
    logProp() {
        // Use inner name C to refer to class
        console.log(`logProp: ${C.prop}`);
    }
}
C.prop = 'Hi!';

const D = C;
C = null;

// C is not a class, anymore:
new C().logProp();
    // TypeError: C is not a function

// But inside the class, the identifier C
// still works
new D().logProp();
    // constructor: Hi!
    // logProp: Hi!
```

## Prototype chains

The previous example creates the following objects.



*Prototype chains* are objects linked via the [[Prototype]] relationship (which is an inheritance relationship). In the diagram, you can see two prototype chains:

## Allocating and initializing instances

The data flow between class constructors is different from the canonical way of subclassing in ES5. Under the hood, it roughly looks as follows.

```
// Base class: this is where the instance is allocated
function Person(name) {
   // Performed before entering this constructor:
   this = Object.create(new.target.prototype);

   this.name = name;
}
...

function Employee(name, title) {
   // Performed before entering this constructor:
   this = uninitialized;

   this = Reflect.construct(Person, [name], new.target); // (A)
      // super(name);

   this.title = title;
}
Object.setPrototypeOf(Employee, Person);
…

const jane = Reflect.construct( // (B)
            Employee, ['Jane', 'CTO'],
            Employee);
   // const jane = new Employee('Jane', 'CTO')
```

The instance object is created in different locations in ES6 and ES5:

- In ES6, it is created in the base constructor, the last in a chain of constructor calls. The superconstructor is invoked via super(), which triggers a constructor call.
- In ES5, it is created in the operand of new, the first in a chain of constructor calls. The superconstructor is invoked via a function call

The previous code uses two new ES6 features:

- new.target is an implicit parameter that all functions have. In a chain of constructor calls, its role is similar to this in a chain of supermethod calls.
    - If a constructor is directly invoked via new (as in line B), the value of new.target is that constructor.
    - If a constructor is called via super() (as in line A), the value of new.target is the new.target of the constructor that makes the call.
    - During a normal function call, it is undefined. That means that you can use new.target to determine whether a function was function-called or constructor-called (via new).
    - Inside an arrow function, new.target refers to the new.target of the surrounding non-arrow function.
- Reflect.construct() lets you make constructor calls while specifying new.target via the last parameter.

The advantage of this way of subclassing is that it enables normal code to subclass built-in constructors (such as Error and Array). A later section explains why a different approach was necessary.

**As a reminder, here is how you do subclassing in ES5:**

```
function Person(name) {
   this.name = name;
}
…

function Employee(name, title) {
   Person.call(this, name);
   this.title = title;
}
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;
```

## The extends clause

The value of an extends clause must be “constructible” (invocable via new). null is allowed, though.

```
class C {

}
```

- Constructor kind: base
- Prototype of C: Function.prototype (like a normal function)
- Prototype of C.prototype: Object.prototype (which is also the prototype of objects created via object literals)

```
class C extends B {

}
```

- Constructor kind: derived
- Prototype of C: B
- Prototype of C.prototype: B.prototype

```
class C extends Object {

}
```

- Constructor kind: derived
- Prototype of C: Object
- Prototype of C.prototype: Object.prototype

Note the following subtle difference with the first case: If there is no extends clause, the class is a base class and allocates instances. If a class extends Object, it is a derived class and Object allocates the instances. The resulting instances (including their prototype chains) are the same, but you get there differently.

```
class C extends null {

}
```

- Constructor kind: base (as of ES2016)
- Prototype of C: Function.prototype
- Prototype of C.prototype: null

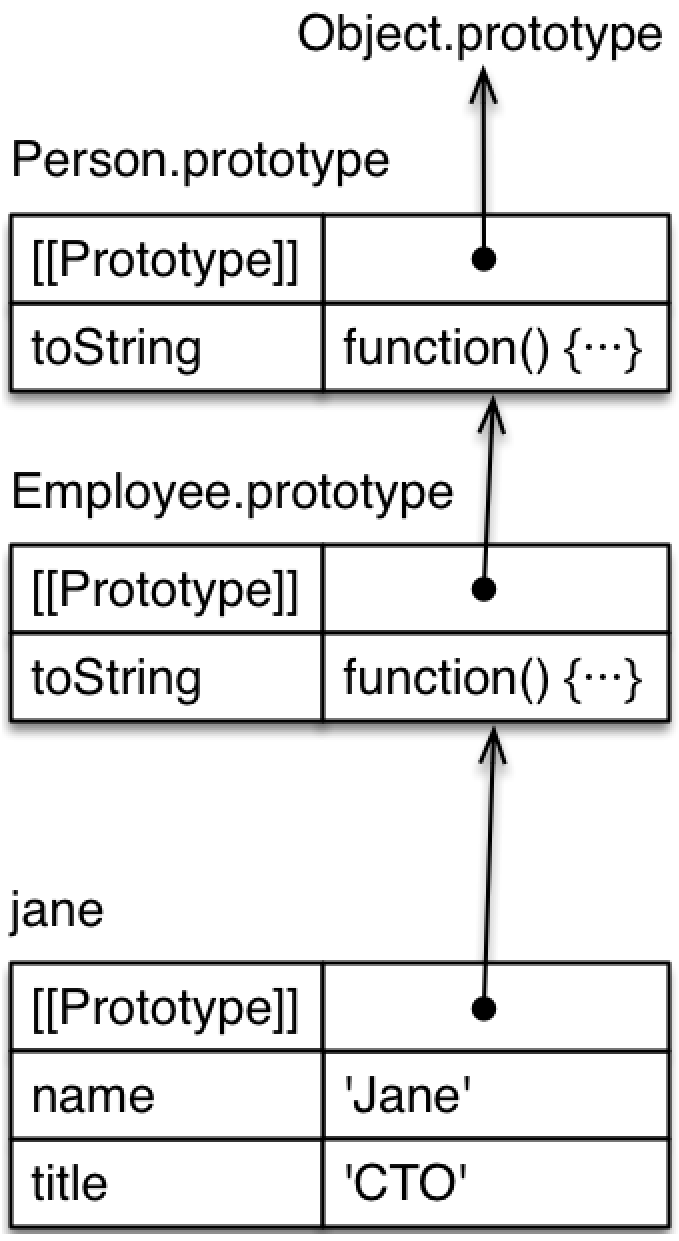
Such a class lets you avoid Object.prototype in the prototype chain.

## Referring to superproperties in methods

The following ES6 code makes a supermethod call in line B.

```
class Person {
    constructor(name) {
        this.name = name;
    }
    toString() { // (A)
        return `Person named ${this.name}`;
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    toString() {
        return `${super.toString()} (${this.title})`; // (B)
    }
}

const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)
```

To understand how super-calls work, let’s look at the object diagram of jane:



## Symbols in ES6

ES6 added Symbol as a new primitive type. Unlike other primitive types such as number, boolean, null, undefined, and string, the symbol type doesn't have a literal form.They are created via a factory function:

```
const mySymbol = Symbol('mySymbol');
```

Every time you call the factory function, a new and unique symbol is created. The optional parameter is a descriptive string that is shown when printing the symbol (it has no other purpose):

```
> mySymbol
```

Symbol(mySymbol);

The Symbol() function creates a new *unique* value each time you call it:

console.log(Symbol() === Symbol()); // false

## Use case 1: unique property keys symbol

Symbols are mainly used as unique property keys – a symbol never clashes with any other property key (symbol or string). For example, you can make an object iterable (usable via the for-of loop and other language mechanisms), by using the symbol stored in Symbol.iterator as the key of a method

```
const iterableObject = {
    [Symbol.iterator]() { // (A)
        ...
    }
}
for (const x of iterableObject) {
    console.log(x);
}
// Output:
// hello
// world
```

In line A, a symbol is used as the key of the method. This unique marker makes the object iterable and enables us to use the for-of loop.

## Use case 2: constants representing concepts

In ECMAScript 5, you may have used strings to represent concepts such as colors. In ES6, you can use symbols and be sure that they are always unique:

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');

function getComplement(color) {
    switch (color) {
        case COLOR_RED:
            return COLOR_GREEN;
        case COLOR_ORANGE:
            return COLOR_BLUE;
        case COLOR_YELLOW:
            return COLOR_VIOLET;
        case COLOR_GREEN:
            return COLOR_RED;
        case COLOR_BLUE:
            return COLOR_ORANGE;
        case COLOR_VIOLET:
            return COLOR_YELLOW;
        default:
            throw new Exception('Unknown color: '+color);
    }
}
```

Every time you call Symbol('Red'), a new symbol is created.
Therefore, COLOR_RED can never be mistaken for another value.
That would be different if it were the string 'Red'.

## Pitfall: you can’t coerce symbols to strings

Coercing (implicitly converting) symbols to strings throws exceptions:

```
const sym = Symbol('desc');

const str1 = '' + sym; // TypeError
const str2 = `${sym}`; // TypeError
```

The only solution is to convert explicitly:

```
const str2 = String(sym); // 'Symbol(desc)'
const str3 = sym.toString(); // 'Symbol(desc)'
```

Forbidding coercion prevents some errors, but also makes working with symbols more complicated.

## Which operations related to property keys are aware of symbols?

The following operations are aware of symbols as property keys:

- Reflect.ownKeys()

- Property access via []
- Object.assign()

The following operations ignore symbols as property keys:

- Object.keys()
- Object.getOwnPropertyNames()
- for-in loop

## Sharing symbols

ES6 provides you with the global symbol registry that allows you to share symbols globally. If you want to create a symbol that will be shared, you use the Symbol.for() method instead of calling the Symbol() function.

The Symbol.for() method accepts a single parameter that can be used for symbol's description as shown in the following example:

let ssn = Symbol.for('ssn');

The Symbol.for() method first searches for the symbol with the  ssn key in the global symbol registry. It returns the existing symbol if there is one. Otherwise, the Symbol.for() method creates a new symbol, registers it to the global symbol registry with the specified key, and returns the symbol.
Later, if you call the Symbol.for() method using the same key, the Symbol.for() method will return the existing symbol.

let citizenID = Symbol.for('ssn');
console.log(ssn === citizenID); // true

To get the key associated with a symbol, you use the Symbol.keyFor() method as shown in the following example:

console.log(Symbol.keyFor(citizenID)); // 'ssn'
If a symbol that does not exist in the global symbol registry, the System.keyFor() method returns **undefined**.

let systemID = Symbol('sys');
console.log(Symbol.keyFor(systemID)); // undefined

## Using symbol as the computed property name of an object

You can use symbols as computed property names. See the following example.
let status = Symbol('status');

```
let task = {
    [status]: statuses.OPEN,
    description: 'Learn ES6 Symbol'
};
console.log(task);
output:
{ description: 'Learn ES6 Symbol', [Symbol(status)]: Symbol(Open) }
```

## Well know symbols:

### Symbol.hasInstance

```
type[Symbol.hasInstance](obj);

class Stack {
}
console.log([] instanceof Stack); // false
```

The [] array is not an instance of the Stack class, therefore, the instanceof operator returns false in this example.

```
class Stack {
    static [Symbol.hasInstance](obj) {
        return Array.isArray(obj);
    }
}
console.log([] instanceof Stack); // true
```

### symbol.iterator

Internally, JavaScript engine first calls the Symbol.iterator method of the numbers array to get the iterator object. Then, it calls the iterator.next() method and copies the value property fo the iterator object into the num variable. After three iterations, the done property of the result object is true, the loop exits.

```
var numbers = [1, 2, 3];
for (let num of numbers) {
    console.log(num);
}

// 1
// 2
// 3
```

You can access the default iterator object via System.iterator symbol as follows:

```
var numbers = [1, 2, 3];
var iterator = numbers[Symbol.iterator]();

console.log(iterator.next()); // Object {value: 1, done: false}
console.log(iterator.next()); // Object {value: 2, done: false}
console.log(iterator.next()); // Object {value: 3, done: false}
console.log(iterator.next()); // Object {value: undefined, done: true}
```

## Symbol.toPrimitive

The Symbol.toPrimitive method determines what should happen when an object is converted into a primitive value. The JavaScript engine defines the Symbol.toPrimitive method on the prototype of each standard type.

The Symbol.toPrimitive method takes a hint argument which has one of three values: “number”, “string”, and “default”. The hint argument specifies the type of the return value. The hint parameter is filled by the JavaScript engine based on the context in which the object is used. Here is an example of using the Symbol.toPrimitive method.

```
function Money(amount, currency) {
   this.amount = amount;
   this.currency = currency;
}
Money.prototype[Symbol.toPrimitive] = function(hint) {
   var result;
   switch (hint) {
      case 'string':
         result = this.amount + this.currency;
         break;
      case 'number':
         result = this.amount;
         break;
      case 'default':
         result = this.amount + this.currency;
         break;
   }
   return result;
}

var price = new Money(799, 'USD');

console.log('Price is ' + price); // Price is 799USD
console.log(+price + 1); // 800
console.log(String(price)); // 799USD
```

## JavaScript Iterators

ES6 introduced a new loop construct called for...of to eliminate the standard loop's complexity and avoid the errors caused by keeping track of loop indexes.
To iterate over the elements of the ranks array, you use the following for...of construct:

```
for(let rank of ranks) {
   console.log(rank);
}
```

The above code also demonstrates the use for.. of loop in JS too.
The for...of is far more elegant than the for loop because it shows the true intent of the code – iterate over an array to access each element in the sequence.
On top of this, the for...of loop has the ability to create a loop over any **iterable** object, not just an array.

Since ES6 provides built-in iterators for the collection types  Array, Set, and Map, you don't have to create iterators for these objects.

If you have a custom type and want to make it iterable so that you can use the for...of loop construct, you need to implement the iteration protocols.

The following code creates a Sequence object that returns a list of numbers in the range of ( start, end) with an interval between subsequent numbers.

```
class Sequence {
   constructor( start = 0, end = Infinity, interval = 1 ) {
      this.start = start;
      this.end = end;
      this.interval = interval;
   }
   [Symbol.iterator]() {
      let counter = 0;
      let nextIndex = this.start;
      return  {
         next: () => {
            if ( nextIndex <= this.end ) {
                    let result = { value: nextIndex,  done: false }
                    nextIndex += this.interval;
                    counter++;
                    return result;
                }
                return { value: counter, done: true };
            }
        }
    }
};

let evenNumbers = new Sequence(2, 10, 2);
for (const num of evenNumbers) {
    console.log(num);
}
```

## new primitive type

ECMAScript 6 introduces a new primitive type: symbols. They are tokens that serve as unique IDs. You create symbols via the factory function Symbol() (which is loosely similar to String returning strings if called as a function):

```
const symbol1 = Symbol();
```

Symbol() has an optional string-valued parameter that lets you give the newly created Symbol a description. That description is used when the symbol is converted to a string (via toString() or String()):

```
> const symbol2 = Symbol('symbol2');
> String(symbol2)
'Symbol(symbol2)'
```

Every symbol returned by Symbol() is unique, every symbol has its own identity:

```
> Symbol() === Symbol()
false
```

You can see that symbols are primitive if you apply the typeof operator to one of them – it will return a new symbol-specific result:

```
> typeof Symbol()
'symbol'
```

## Symbols can be used as property keys:

```
const MY_KEY = Symbol();
const obj = {};

obj[MY_KEY] = 123;
console.log(obj[MY_KEY]); // 123
```

## Using symbols to represent concepts

```
const COLOR_RED    = Symbol('Red');
const COLOR_ORANGE = Symbol('Orange');
const COLOR_YELLOW = Symbol('Yellow');
const COLOR_GREEN  = Symbol('Green');
const COLOR_BLUE   = Symbol('Blue');
const COLOR_VIOLET = Symbol('Violet');
```

## Symbols as keys of non-public properties

```
const _counter = Symbol('counter');
const _action = Symbol('action');
class Countdown {
    constructor(counter, action) {
        this[_counter] = counter;
        this[_action] = action;
    }
    dec() {
        let counter = this[_counter];
        if (counter < 1) return;
        counter--;
        this[_counter] = counter;
        if (counter === 0) {
            this[_action]();
        }
    }
}
```

## New static Array methods

### Array.from(arrayLike, mapFunc?, thisArg?)

Array.from()'s basic functionality is to convert two kinds of values to Arrays:

- Array-like values, which have a property length and indexed elements. Examples include the results of DOM operations such as document.getElementsByClassName().
- Iterable values, whose contents can be retrieved one element at a time. Strings and Arrays are iterable, as are ECMAScript's new data structures Map and Set.

The following is an example of converting an Array-like object to an Array:

```
const arrayLike = { length: 2, 0: 'a', 1: 'b' };

// for-of only works with iterable values
for (const x of arrayLike) { // TypeError
    console.log(x);
}

const arr = Array.from(arrayLike);
for (const x of arr) { // OK, iterable
    console.log(x);
}
// Output:
// a
// b
```

### Mapping via Array.from()

Array.from() is also a convenient alternative to using map() generically:

```
const spans = document.querySelectorAll('span.name');

// map(), generically:
const names1 = Array.prototype.map.call(spans, s => s.textContent);

// Array.from():
const names2 = Array.from(spans, s => s.textContent);
```

In this example, the result of document.querySelectorAll() is again an Array-like object, not an Array, which is why we couldn't invoke map() on it. Previously, we converted the Array-like object to an Array in order to call forEach(). Here, we skipped that intermediate step via a generic method call and via the two-parameter.

## from() in subclasses of Array

```
class MyArray extends Array {
    ...
}
const instanceOfMyArray = MyArray.from(anIterable);
```

1)

```
// from() – determine the result's constructor via the receiver
// (in this case, MyArray)
const instanceOfMyArray = MyArray.from([1, 2, 3], x => x * x);

// map(): the result is always an instance of Array
const instanceOfArray   = [1, 2, 3].map(x => x * x);
```

## ES6 and holes in Arrays

Holes are indices "inside" an Array that have no associated element. In other words: An Array arr is said to have a hole at index i if:

- 0 ≤ i < arr.length
- !(i in arr)

For example: The following Array has a hole at index 1.

```
> const arr = ['a',,'b']
'use strict'
> 0 in arr
true
> 1 in arr
false
> 2 in arr
true
> arr[1]
undefined
```

The following table describes how Array.prototype methods handle holes.

| **Method** | **Holes are** | |
|---|---|---|
| concat | Preserved | ['a',,'b'].concat(['c',,'d']) → ['a',,'b','c',,'d'] |
| copyWithin^ES6 | Preserved | [,'a','b',,].copyWithin(2,0) → [,'a',,'a'] |
| entries^ES6 | Elements | [...[,'a'].entries()] → [[0,undefined], [1,'a']] |
| every | Ignored | [,'a'].every(x => x==='a') → true |
| fill^ES6 | Filled | new Array(3).fill('a') → ['a','a','a'] |

| Method | Holes are | |
|---|---|---|
| filter | Removed | ['a',,'b'].filter(x => true) → ['a','b'] |
| find$^{ES6}$ | Elements | [,'a'].find(x => true) → undefined |
| findIndex$^{ES6}$ | Elements | [,'a'].findIndex(x => true) → 0 |
| forEach | Ignored | [,'a'].forEach((x,i) => log(i)); → 1 |
| indexOf | Ignored | [,'a'].indexOf(undefined) → -1 |
| join | Elements | [,'a',undefined,null].join('#') → '#a##' |
| keys$^{ES6}$ | Elements | [...[,'a'].keys()] → [0,1] |
| lastIndexOf | Ignored | [,'a'].lastIndexOf(undefined) → -1 |
| map | Preserved | [,'a'].map(x => 1) → [,1] |
| pop | Elements | ['a',,].pop() → undefined |
| push | Preserved | new Array(1).push('a') → 2 |
| reduce | Ignored | ['#',,undefined].reduce((x,y)=>x+y) → '#undefined' |
| reduceRight | Ignored | ['#',,undefined].reduceRight((x,y)=>x+y) → 'undefined#' |
| reverse | Preserved | ['a',,'b'].reverse() → ['b',,'a'] |
| shift | Elements | [,'a'].shift() → undefined |
| slice | Preserved | [,'a'].slice(0,1) → [,] |
| some | Ignored | [,'a'].some(x => x !== 'a') → false |
| sort | Preserved | [,undefined,'a'].sort() → ['a',undefined,,] |
| splice | Preserved | ['a',,].splice(1,1) → [,] |
| toString | Elements | [,'a',undefined,null].toString() → ',a,,' |
| unshift | Preserved | [,'a'].unshift('b') → 3 |
| values$^{ES6}$ | Elements | [...[,'a'].values()] → [undefined,'a'] |

Notes:

- ES6 methods are marked via the superscript “ES6”.
- JavaScript ignores a trailing comma in an Array literal: ['a',,].length → 2
- Helper function used in the table: const log = console.log.bind(console);

## Maps

The keys of a Map can be arbitrary values:

```
> const map = new Map(); // create an empty Map
> const KEY = {};

> map.set(KEY, 123);
> map.get(KEY)
123
> map.has(KEY)
true
> map.delete(KEY);
true
```

```
> map.has(KEY)
false
```

You can use an Array (or any iterable) with [key, value] pairs to set up the initial data in the Map:

```
const map = new Map([
    [ 1, 'one' ],
    [ 2, 'two' ],
    [ 3, 'three' ], // trailing comma is ignored
]);
```

## Sets

A Set is a collection of unique elements:

```
const arr = [5, 1, 5, 7, 7, 5];

const unique = [...new Set(arr)]; // [ 5, 1, 7 ]
```

As you can see, you can initialize a Set with elements if you hand the constructor an iterable (arr in the example) over those elements.

## WeakMaps

A WeakMap is a Map that doesn’t prevent its keys from being garbage-collected. That means that you can associate data with objects without having to worry about memory leaks. For example:

```
//----- Manage listeners

const _objToListeners = new WeakMap();

function addListener(obj, listener) {
    if (! _objToListeners.has(obj)) {
        _objToListeners.set(obj, new Set());
    }
    _objToListeners.get(obj).add(listener);
}

function triggerListeners(obj) {
    const listeners = _objToListeners.get(obj);
    if (listeners) {
        for (const listener of listeners) {
            listener();
        }
    }
}

//----- Example: attach listeners to an object

const obj = {};
addListener(obj, () => console.log('hello'));
addListener(obj, () => console.log('world'));

//----- Example: trigger listeners

triggerListeners(obj);

// Output:
// hello
// world
```

## JavaScript new.target Metaproperty

new.target metaproperty that detects whether a function or constructor was called using the new operator.

ES6 provides a *metaproperty* named new.target that allows you to detect whether a function or constructor was called using the new operator.

The new.target is available for all functions. However, in arrow functions, the new.target is the one that belongs to the surrounding function.

- The new.target is very useful to inspect at runtime whether a function is being executed as a function or as a constructor.
- It is also handy to determine a specific derived class that was called by using the new operator from within a base class.

## new.target in functions

whether a function was called using the new operator, you use the new.target.

```
function Person(name) {
    if (!new.target) {
        throw "must use new operator with Person";
    }
    this.name = name;
}
```

Now, the only way to use Person () is to instantiate an object from it by using the new operator. If you try to call it as a normal function, you will get an error.

## new.target in constructors

In a class constructor, the new.target refers to the constructor that was invoked directly by the new operator. It is true if the constructor is in the base class and was delegated from a derived constructor. Here is an example.

```
class Person {
    constructor(name) {
        this.name = name;
        console.log(new.target.name);
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
}

let john = new Person('John Doe'); // Person
let lily = new Employee('Lily Bush', 'Programmer'); // Employee
```

Here we have learned, how to use the JavaScript new.target metaproperty to detect whether a function or constructor was called using the new operator.

## Module basics

An ES6 module is a file containing JS code. There's no special module keyword; a module mostly reads just like a script. There are two differences.

- ES6 modules are automatically strict-mode code, even if you don't write "use strict"; in them.
- You can use import and export in modules.

```
// greeting.js
export let message = 'Hi';

export function setMessage(msg) {
  message = msg;
}
// app.js
import {message, setMessage } from './greeting.js';
console.log(message); // 'Hi'
```

```
setMessage('Hello');
console.log(message); // 'Hello'
```

Roughly speaking, when you tell the JS engine to run a module, it has to behave as though these four steps are happening:

1. Parsing: The implementation reads the source code of the module and checks for syntax errors.
2. Loading: The implementation loads all imported modules (recursively). This is the part that isn't standardized yet.
3. Linking: For each newly loaded module, the implementation creates a module scope and fills it with all the bindings declared in that module, including things imported from other modules.

This is the part where if you try to `import {cake} from "paleo"`, but the "paleo" module doesn't actually export anything named `cake`, you'll get an error. And that's too bad, because you were *so close* to actually running some JS code. And having cake!

4. Run time: Finally, the implementation runs the statements in the body of each newly-loaded module. By this time, `import` processing is already finished, so when execution reaches a line of code where there's an `import` declaration… nothing happens!

There's a cool trick. Because the system doesn't specify how loading works, and because you can figure out all the dependencies ahead of time by looking at the `import` declarations in the source code, an implementation of ES6 is free to do all the work at compile time and bundle all your modules into a single file to ship them over the network!

## Static vs. dynamic, or: rules and how to break them

For a dynamic language, JavaScript has gotten itself a surprisingly static module system.

- All flavors of `import` and `export` are allowed only at toplevel in a module. There are no conditional imports or exports, and you can't use `import` in function scope.
- All exported identifiers must be explicitly exported by name in the source code. You can't programmatically loop through an array and export a bunch of names in a data-driven way.
- Module objects are frozen. There is no way to hack a new feature into a module object, polyfill style.
- *All* of a module's dependencies must be loaded, parsed, and linked eagerly, before any module code runs. There's no syntax for an `import` that can be loaded lazily, on demand.

- There is no error recovery for import errors. An app may have hundreds of modules in it, and if anything fails to load or link, nothing runs. You can't import in a try/catch block. (The upside here is that because the system is so static, webpack can detect those errors for you at compile time.)
- There is no hook allowing a module to run some code before its dependencies load. This means that modules have no control over how their dependencies are loaded.

## When can I use ES6 modules?

To use modules today, you'll need a compiler such as Traceur or Babel. Earlier in this series, Gastón I. Silva showed how to use Babel and Broccoli to compile ES6 code for the web; building on that article, Gastón has a working example with support for ES6 modules. This post by Axel Rauschmayer contains an example using Babel and webpack.

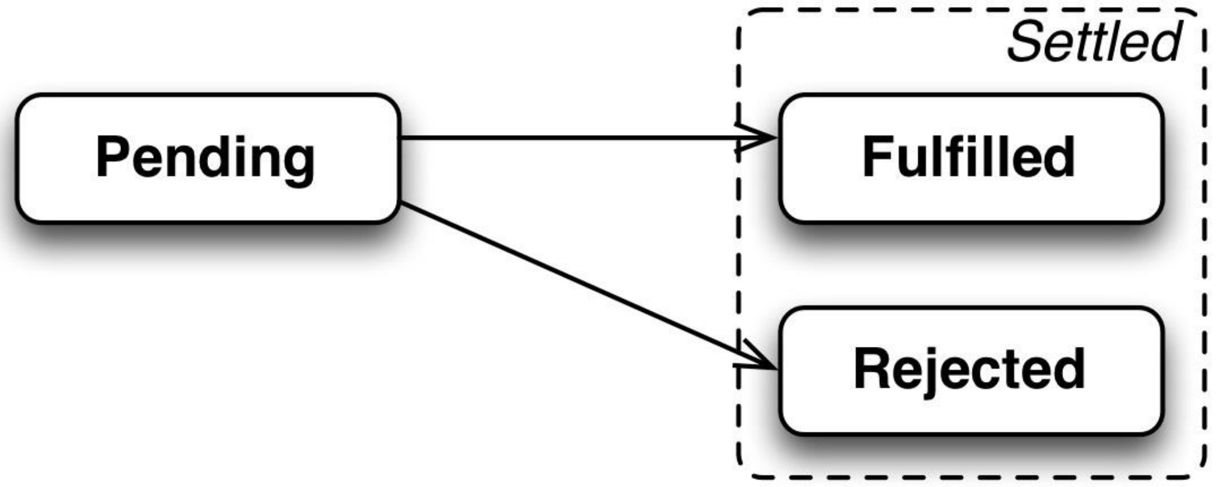
The ES6 module system was designed mainly by Dave Herman and Sam Tobin-Hochstadt, who defended the static parts of the system against all comers (including me) through years of controversy. Jon Coppeard is implementing modules in Firefox. Additional work on a JavaScript Loader Standard is underway.

## The states of Promises

Each Promise is always in either one of three (mutually exclusive) states:

- Pending: the result hasn't been computed, yet (the initial state of each Promise)
- Fulfilled: the result was computed successfully
- Rejected: a failure occurred during computation

A Promise is *settled* (the computation it represents has finished) if it is either fulfilled or rejected. A Promise can only be settled once and then stays settled. Subsequent attempts to settle have no effect.



The parameter of new Promise() (starting in line A) is called an *executor*:

- Resolving: If the computation went well, the executor sends the result via resolve(). That usually fulfills the Promise p. But it may not – resolving with a Promise q leads to p tracking q: If q is still pending then so is p. However q is settled, p will be settled the same way.
- Rejecting: If an error happened, the executor notifies the Promise consumer via reject(). That always rejects the Promise.

If an exception is thrown inside the executor, p is rejected with that exception.

## Consuming a Promise

As a consumer of promise, you are notified of a fulfillment or a rejection via *reactions* – callbacks that you register with the methods then() and catch():

```
promise
.then(value => { /* fulfillment */ })
.catch(error => { /* rejection */ });
```

What makes Promises so useful for asynchronous functions (with one-off results) is that once a Promise is settled, it doesn't change anymore. Furthermore, there are never any race conditions, because it doesn't matter whether you invoke then() or catch() before or after a Promise is settled:

- Reactions that are registered with a Promise before it is settled, are notified of the settlement once it happens.
- Reactions that are registered with a Promise after it is settled, receive the cached settled value "immediately" (their invocations are queued as tasks).

Note that catch() is simply a more convenient (and recommended) alternative to calling then(). That is, the following two invocations are equivalent:

```
promise.then(
   null,
   error => { /* rejection */ });

promise.catch(
   error => { /* rejection */ });
```

## Promises are always asynchronous

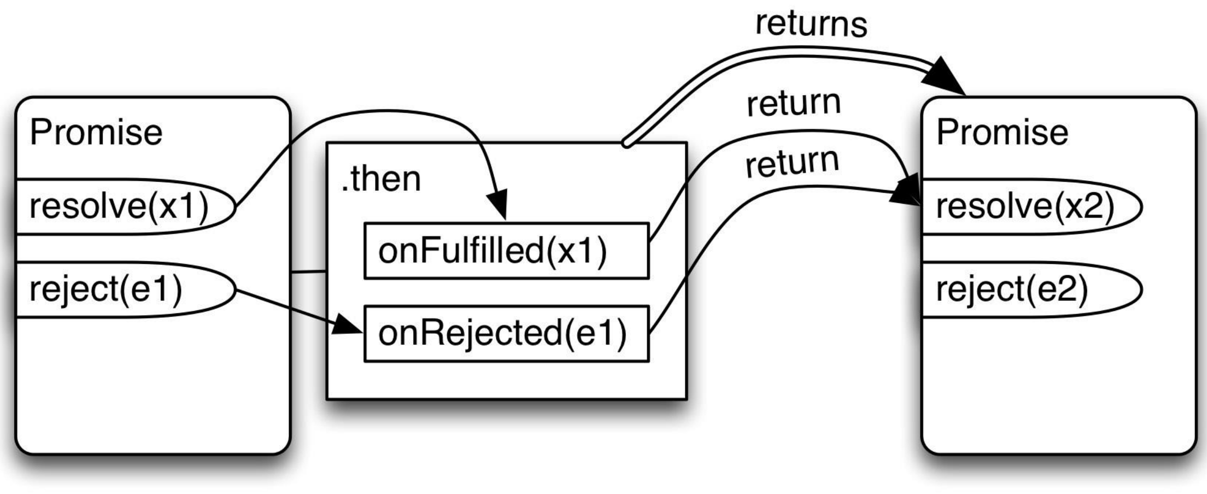
A Promise library has complete control over whether results are delivered to Promise reactions synchronously (right away) or asynchronously (after the current continuation, the current piece of code, is finished).

## Promise Chaining

The next feature we implement is chaining:

- then() returns a Promise that is resolved with what either onFulfilled or onRejected return.
- If onFulfilled or onRejected are missing, whatever they would have received is passed on to the Promise returned by then().

```
.then(onFulfilled, onRejected) {
    const returnValue = new Promise(); // (A)
    const self = this;

    let fulfilledTask;
    if (typeof onFulfilled === 'function') {
        fulfilledTask = function () {
            const r = onFulfilled(self.promiseResult);
            returnValue.resolve(r); // (B)
        };
    } else {
        fulfilledTask = function () {
            returnValue.resolve(self.promiseResult); // (C)
        };
    }

    let rejectedTask;
    if (typeof onRejected === 'function') {
        rejectedTask = function () {
            const r = onRejected(self.promiseResult);
            returnValue.resolve(r); // (D)
        };
    } else {
        rejectedTask = function () {
            // `onRejected` has not been provided
            // => we must pass on the rejection
            returnValue.reject(self.promiseResult); // (E)
        };
    }
    …
    return returnValue; // (F)
}
```



## Promise.prototype.finally()

.finally() works as follows:

```
Promise
.then(result => {···})
.catch(error => {···})
.finally(() => {···});
```

finally's callback is always executed. Compare:

- then's callback is only executed if promise is fulfilled.
- catch's callback is only executed if promise is rejected. Or if then's callback throws an exception or returns a rejected Promise.

In other words: Take the following piece of code.

```
promise
.finally(() => {
  «statements»
```

## Use case of Promise

For example:

```
let connection;
db.open()
.then(conn => {
    connection = conn;
    return connection.select({ name: 'Jane' });

.then(result => {
    // Process result
    // Use `connection` to make more queries
})
…
.catch(error => {
    // handle errors
})
.finally(() => {
    connection.close();
```

## What are generators?

*Generators* are functions that can be paused and resumed (think cooperative multitasking or coroutines), which enables a variety of applications.

You can think of generators as processes (pieces of code) that you can pause and resume:

```
function* genFunc() {
    // (A)
    console.log('First');
    yield;
    console.log('Second');
}
```

yield is an operator with which a generator can pause itself. Additionally, generators can also receive input and send output via yield.

- First, you see the asterisk (*) after the function keyword. The asterisk denotes that the generate() is a generator, not a normal function.
- Second, the yield statement returns a value and pauses the execution of the function

When you call a generator function genFunc(), you get a *generator object* genObj that you can use to control the process:

```
const genObj = genFunc();
```

The process is initially paused in line A. genObj.next() resumes execution, a yield inside genFunc() pauses execution:

```
genObj.next();
// Output: First
genObj.next();
// output: Second
```

## Kinds of generators

**There are four kinds of generators:**

1. Generator function declarations:
```
function* genFunc() { ··· }
const genObj = genFunc();
```
2. Generator function expressions:
```
const genFunc = function* () { ··· };
const genObj = genFunc();
```
3. Generator method definitions in object literals:
```
const obj = {
    * generatorMethod() {
        ···
    }
};
const genObj = obj.generatorMethod();
```
4. Generator method definitions in class definitions (class declarations or class expressions):
```
class MyClass {
    * generatorMethod() {
        ···
    }
}
const myInst = new MyClass();
const genObj = myInst.generatorMethod();
```

## Use case: implementing iterables

The objects returned by generators are iterable; each yield contributes to the sequence of iterated values. Therefore, you can use generators to implement iterables, which can be consumed by various ES6 language mechanisms: for-of loop, spread operator (...), etc.

The following function returns an iterable over the properties of an object, one [key, value] pair per property:

```
function* objectEntries(obj) {
const propKeys = Reflect.ownKeys(obj);
  for (const propKey of propKeys) {
      // `yield` returns a value and then pauses
      // the generator. Later, execution continues
      // where it was previously paused.
      yield [propKey, obj[propKey]];
   }
}
```

objectEntries() is used like this:

```
const jane = { first: 'Jane', last: 'Doe' };
for (const [key,value] of objectEntries(jane)) {
   console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

## Throwing an exception from a generator

If an exception leaves the body of a generator then next() throws it:

```
function* genFunc() {
   throw new Error('Problem!');
}
const genObj = genFunc();
genObj.next(); // Error: Problem!
```

## Returning from a generator

An implicit return is equivalent to returning undefined. Let's examine a generator with an explicit return:

```
function* genFuncWithReturn() {
   yield 'a';
   yield 'b';
   return 'result';
}
```

The returned value shows up in the last object returned by next(), whose property done is true:

```
> const genObjWithReturn = genFuncWithReturn();
> genObjWithReturn.next()
{ value: 'a', done: false }
> genObjWithReturn.next()
{ value: 'b', done: false }
> genObjWithReturn.next()
{ value: 'result', done: true }
```

However, most constructs that work with iterables ignore the value inside the done object:

```
for (const x of genFuncWithReturn()) {
    console.log(x);
}
// Output:
// a
// b

const arr = [...genFuncWithReturn()]; // ['a', 'b']
```

yield*, an operator for making recursive generator calls, does consider values inside done objects.

## Example: iterating over properties

Let’s look at an example that demonstrates how convenient generators are for implementing iterables. The following function, objectEntries(), returns an iterable over the properties of an object:

```
function* objectEntries(obj) {
    // In ES6, you can use strings or symbols as property keys,
    // Reflect.ownKeys() retrieves both
    const propKeys = Reflect.ownKeys(obj);

    for (const propKey of propKeys) {
        yield [propKey, obj[propKey]];
    }
}
```

This function enables you to iterate over the properties of an object jane via the for-of loop:

```
const jane = { first: 'Jane', last: 'Doe' };
for (const [key,value] of objectEntries(jane)) {
    console.log(`${key}: ${value}`);
}
// Output:
// first: Jane
// last: Doe
```

## Why use the keyword function* for generators and not generator?

Due to backward compatibility, using the keyword generator wasn’t an option. For example, the following code (a hypothetical ES6 anonymous generator expression) could be an ES5 function call followed by a code block.

```
generator (a, b, c) {
    ···
}
```

I find that the asterisk naming scheme extends nicely to yield*.

- Generators are created by the generator function function* f(){}.
- Generators do not execute its body immediately when they are invoked.
- Generators can pause midway and resumes their executions where they were paused. The yield statement pauses the execution of a generator and returns a value.
- Generators are iterable so you can use them with the for...of loop.

## yield*: the full story

The yield keyword allows you to pause and resume a generator function (function*).

yield is only a reserved word in strict mode. A trick is used to bring it to ES6 sloppy mode: it becomes a *contextual keyword*, one that is only available inside generators.

As a rough rule of thumb, yield* performs (the equivalent of) a function call from one generator (the *caller*) to another generator (the *callee*).

yield propagates yielded values from the callee to the caller. Now that we are interested in generators receiving input, another aspect becomes relevant: yield* also forwards input received by the caller to the callee. In a way, the callee becomes the active generator and can be controlled via the caller's generator object.

The following generator function caller() invokes the generator function callee() via yield*.

```
function* callee() {
    console.log('callee: ' + (yield));
}
function* caller() {
    while (true) {
        yield* callee();
    }
}
```

callee logs values received via next(), which allows us to check whether it receives the value 'a' and 'b' that we send to caller.

```
> const callerObj = caller();

> callerObj.next() // start
{ value: undefined, done: false }

> callerObj.next('a')
```

```
callee: a
{ value: undefined, done: false }

> callerObj.next('b')
callee: b
{ value: undefined, done: false }
```
throw() and return() are forwarded in a similar manner.

```
function* foo() {
    yield 1;
    yield 2;
    yield 3;
}

let f = foo();

console.log(f.next());
```
As you can see the value that follows the yield is added to the value property of the return object when the next() is called.

## Using the yield to return individual elements of an array

See the following generator function:

```
function* yieldArrayElements() {
    yield 1;
    yield* [ 20, 30, 40 ];
}

let a = yieldArrayElements();

console.log(a.next()); // { value: 1, done: false }
console.log(a.next()); // { value: 20, done: false }
console.log(a.next()); // { value: 30, done: false }
console.log(a.next()); // { value: 40, done: false }
```

## Reflect

The global object Reflect implements all interceptable operations of the JavaScript meta object protocol as methods. The names of those methods are the same as those of the handler methods, which, as we have seen, helps with forwarding operations from the handler to the target.

- Reflect.apply(target, thisArgument, argumentsList): any
  Similar to Function.prototype.apply().

- Reflect.construct(target, argumentsList, newTarget=target): object
  The new operator as a function. target is the constructor to invoke, the optional parameter newTarget points to the constructor that started the current chain of constructor calls.
- Reflect.defineProperty(target, propertyKey, propDesc): boolean
  Similar to Object.defineProperty().
- Reflect.deleteProperty(target, propertyKey): boolean
  The delete operator as a function. It works slightly differently, though: It returns true if it successfully deleted the property or if the property never existed. It returns false if the property could not be deleted and still exists. The only way to protect properties from deletion is by making them non-configurable. In sloppy mode, the delete operator returns the same results. But in strict mode, it throws a TypeError instead of returning false.
- Reflect.get(target, propertyKey, receiver=target): any
  A function that gets properties. The optional parameter receiver points to the object where the getting started. It is needed when get reaches a getter later in the prototype chain. Then it provides the value for this.
- Reflect.getOwnPropertyDescriptor(target, propertyKey): undefined|PropDesc
  Same as Object.getOwnPropertyDescriptor().
- Reflect.getPrototypeOf(target): null|object
  Same as Object.getPrototypeOf().
- Reflect.has(target, propertyKey): boolean
  The in operator as a function.
- Reflect.isExtensible(target): boolean
  Same as Object.isExtensible().
- Reflect.ownKeys(target): Array<PropertyKey>
  Returns all own property keys in an Array: the string keys and symbol keys of all own enumerable and non-enumerable properties.
- Reflect.preventExtensions(target): boolean
  Similar to Object.preventExtensions().
- Reflect.set(target, propertyKey, value, receiver=target): boolean
  A function that sets properties.
- Reflect.setPrototypeOf(target, proto): boolean
  The new standard way of setting the prototype of an object. The current non-standard way, that works in most engines, is to set the special property __proto__.

### Object.* versus Reflect.*

Going forward, Object will host operations that are of interest to normal applications, while Reflect will host operations that are more low-level.

## Reflection in Javascript

reflection is the ability of a program to manipulate variables, properties, and methods of objects at runtime.

Prior to ES6, JavaScript already has reflection features even though they were not officially called that by the community or the specification. For example, methods like Object.keys(), Object.getOwnPropertyDescriptor(), and Array.isArray() are the classic reflection features.

ES6 introduces a new global object called Reflect that allows you to call methods, construct objects, get and set properties, manipulate and extend properties.

The Reflect API is important because it allows you to develop programs and frameworks that are able to handle dynamic code.

Javascript has built-in support for introspection and self-modification. These features are provided as part of the language, rather than through a distinct meta object protocol. This is largely because Javascript objects are represented as flexible records mapping strings to values.

Property names can be computed at runtime and their value can be retrieved using array indexing notation. The following code snippet demonstrates introspection and self-modification:

```
var o = {
x : 5 ,
m : function ( a ) { . . . }
};

          / / Introspection :
          o [ ” x ” ]                      / / computed property access
          ” x ” in o                      / / property lookup
          for ( prop in o ) { . . . }     / / property enumeration
          o [ ” m ” ] . apply ( o , [ 4 2 ] )  / / reflective method call
          / / s e l f −m o d i f i c a t i o n :
          o [ ” x ” ] = 6                / / computed property assignment
          o . z = 7                      / / add property
          delete o . z                   / / remove a property
```

The Reflect object is the dual of a Proxy handler object: a proxy handler can uniformly intercept operations on an object, while the Reflect object can uniformly perform these operations on an object. The following code snippet illustrates this duality for the property query operation:

```
var proxy = Proxy ( target , handler ) ;
name in proxy                     / / equivalent to : handler . has ( target , name )
Reflect . has ( target , name )   / equivalent to :  name in target
Reflect . has ( proxy , name )    / / equivalent to : name in p roxy
                                  / / and thus to : handle r . has ( target , name )
```

## Creating objects: Reflect.construct()

The Reflect.construct() method behaves like the new operator, but as a function. It is equivalent to calling the new target(...args) with the possibility to specify a different prototype:

```
Reflect.construct(target, args [, newTarget])
```

The Reflect.construct() returns the new instance of the target, or the newTarget if specified, initialized by the target as a constructor with the given array-like object args. See the following example:

```
class Person {
   constructor(firstName, lastName) {
      this.firstName = firstName;
      this.lastName = lastName;
   }
   get fullName() {
      return `${this.firstName} ${this.lastName}`;
   }
};

let args = ['John', 'Doe'];

let john = Reflect.construct(
   Person,
   args
);

console.log(john instanceof Person);
console.log(john.fullName); // John Doe
```

The Reflect.apply() provides the same feature as the Function.prototype.apply() but less verbose and easier to understand:

```
let result = Reflect.apply(Math.max, Math, [10, 20, 30]);
console.log(result);
```

Here is the syntax of the Reflect.apply() method:

```
Reflect.apply(target, thisArg, args)
```

## Defining a property: Reflect.defineProperty()

The Reflect.defineProperty() is like the Object.defineProperty(). However, it returns a Boolean indicating whether or not the property was defined successfully instead of throwing an exception:

```
Reflect.defineProperty(target, propertyName, propertyDescriptor)
```

See the following example:

```
let person = {
    name: 'John Doe'
};

if (Reflect.defineProperty(person, 'age', {
        writable: true,
        configurable: true,
        enumerable: false,
        value: 25,
    })) {
    console.log(person.age);
} else {
    console.log('Cannot define the age property on the person object.');

}
```

## Reflect API

Unlike the most global objects, the Reflect is not a constructor. It means that you cannot use Reflect with the new operator or invoke the Reflect as a function. It is similar to the Math and JSON objects. All the methods of the Reflect object are static.

- Reflect.apply() – call a function with specified arguments.
- Reflect.construct() – act like the new operator, but as a function. It is equivalent to calling new target(...args).
- Reflect.defineProperty() – is similar to Object.defineProperty(), but return a Boolean value indicating whether or not the property was successfully defined on the object.
- Reflect.deleteProperty() – behave like the delete operator, but as a function. It's equivalent to calling the delete objectName[propertyName].
- Reflect.get() – return the value of a property.
- Reflect.getOwnPropertyDescriptor() – is similar to Object.getOwnPropertyDescriptor(). It returns a property descriptor of a property if the property exists on the object, or undefined otherwise.
- Reflect.getPrototypeOf() – is the same as Object.getPrototypeOf().
- Reflect.has() – work like the in operator, but as a function. It returns a boolean indicating whether an property (either owned or inherited) exists.
  Reflect.isExtensible() – is the same as Object.isExtensible().
- Reflect.ownKeys() – return an array of the owned property keys (not inherited) of an object.
- Reflect.preventExtensions() – is similar to Object.preventExtensions(). It returns a Boolean.
- Reflect.set() – assign a value to a property and return a Boolean value which is true if the property is set successfully.

- Reflect.setPrototypeOf() – set the prototype of an object.

## Proxies

A JavaScript Proxy is an object that wraps another object (target) and intercepts the fundamental operations of the target object.

The fundamental operations can be the property lookup, assignment, enumeration, and function invocations, etc.

Generic wrappers. Proxies that wrap other objects in the same address space. Example uses include access control wrappers (e.g. revokable references), higher-order contracts [Findler and Felleisen 2002], profiling, taint tracking, etc. Virtual objects.

Proxies that emulate other objects, without the emulated objects having to be present in the same address space. Examples include remote object proxies (emulate objects in other address spaces), persistent objects (emulate objects stored in databases), transparent futures (emulate objects not yet computed), lazily instantiated objects, test mock-ups, etc.

Our Proxy API supports intercession by means of distinct proxy objects. The behavior of a proxy object is controlled by a separate handler object. The methods of the handler object are traps that are called whenever a corresponding operation is applied to the proxy object. Handlers are effectively “meta-objects” and their interface effectively defines a “metaobject protocol”. A proxy object is created as follows:

```
var proxy = Proxy ( target , handler ) ;
```

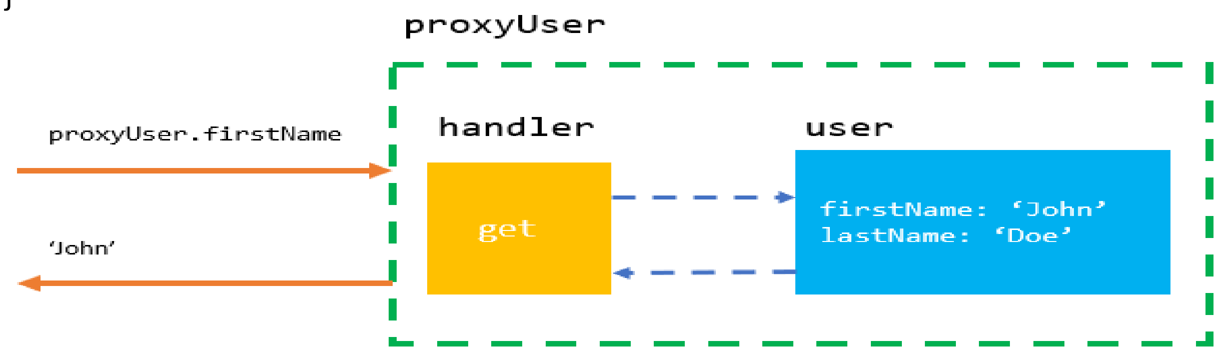
In this syntax:

- target – is an object to wrap.
- handler – is an object that contains methods to control the behaviors of the target. The methods inside the handler object are called traps.

```
const user = {
    firstName: 'John',
    lastName: 'Doe',
    email: 'john.doe@example.com',
}
```

Second, define a handler object:

```
const handler = {
    get(target, property) {
        console.log(`Property ${property} has been read.`);
        return target[property];
    }
}
```



Here, target is an existing Javascript object that is going to be wrapped by the newborn proxy. handler is an object that may implement a particular meta-level API. Below Figure depicts the relationship between these objects.

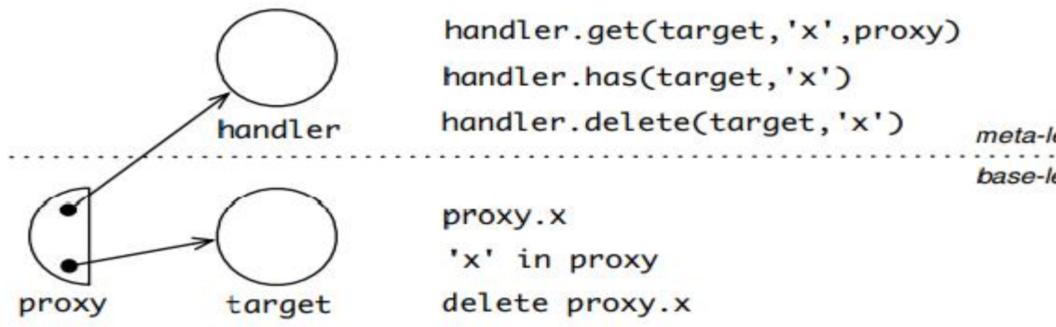
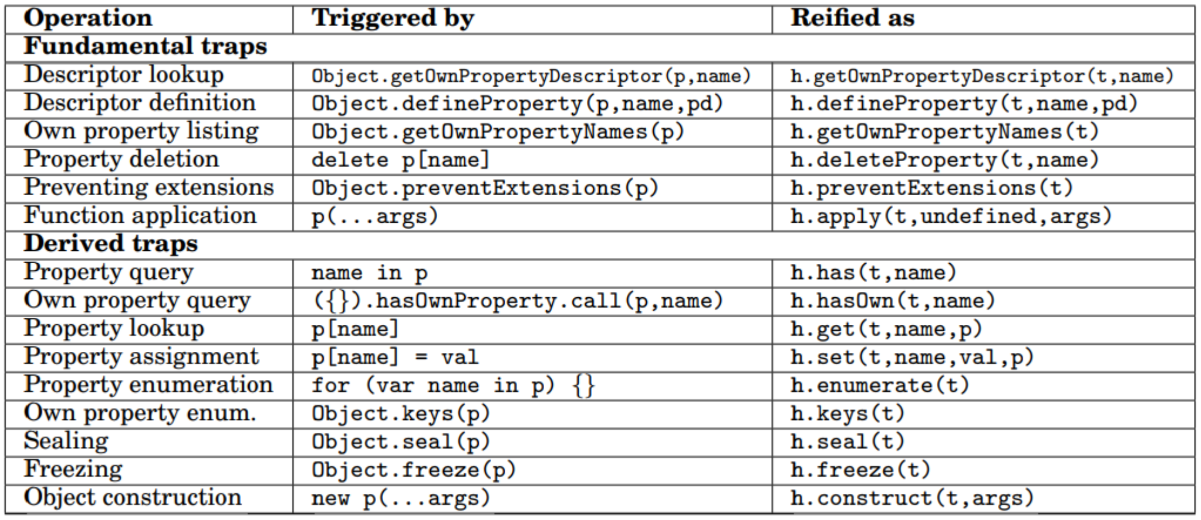


Fig. Relationship between proxy, target and handler

Table I lists those base-level operations applicable to objects that can be trapped by handlers. An intercepted operation mostly simply triggers the corresponding trap and returns its result. The enumerate trap must return an array of strings representing the enumerable property names of the proxy. The corresponding for-in loop is then driven by

Table I. Operations reified on `p = Proxy(t,h)`



| Operation | Triggered by | Reified as |
|---|---|---|
| **Fundamental traps** | | |
| Descriptor lookup | `Object.getOwnPropertyDescriptor(p,name)` | `h.getOwnPropertyDescriptor(t,name)` |
| Descriptor definition | `Object.defineProperty(p,name,pd)` | `h.defineProperty(t,name,pd)` |
| Own property listing | `Object.getOwnPropertyNames(p)` | `h.getOwnPropertyNames(t)` |
| Property deletion | `delete p[name]` | `h.deleteProperty(t,name)` |
| Preventing extensions | `Object.preventExtensions(p)` | `h.preventExtensions(t)` |
| Function application | `p(...args)` | `h.apply(t,undefined,args)` |
| **Derived traps** | | |
| Property query | `name in p` | `h.has(t,name)` |
| Own property query | `({}).hasOwnProperty.call(p,name)` | `h.hasOwn(t,name)` |
| Property lookup | `p[name]` | `h.get(t,name,p)` |
| Property assignment | `p[name] = val` | `h.set(t,name,val,p)` |
| Property enumeration | `for (var name in p) {}` | `h.enumerate(t)` |
| Own property enum. | `Object.keys(p)` | `h.keys(t)` |
| Sealing | `Object.seal(p)` | `h.seal(t)` |
| Freezing | `Object.freeze(p)` | `h.freeze(t)` |
| Object construction | `new p(...args)` | `h.construct(t,args)` |

Because Javascript methods are just functions, a method invocation proxy.m(a,b) is reified as a property access handler.get(target,"m",proxy) that is expected to return a function. That function is immediately applied to the arguments [a,b] with its this-pseudovariable bound to proxy

All traps in the above API are optional. If a handler does not define a trap, the proxy will forward the intercepted operation to its target unmodified. For instance, if handler does not define a get trap, then proxy["x"] is equivalent to target["x"]. The distinction between proxy objects and regular objects ensures that non-proxy objects (which we expect make up the vast majority of objects in a typical heap) do not pay the runtime costs associated with intercession

Finally, the references that a proxy holds to its target and handler are immutable and inaccessible to clients of the proxy. As an illustration of our API, consider a proxy wrapper that simply wants to log all property assignments performed on its wrapped target object, but otherwise does not want to change the behavior of the wrapped object:

```
    function makeChangeLogger ( target , log )
    {
                            return Proxy ( target , {
                set : function ( target , name , value , receiver ) {
                var success = Reflect . set ( target , name , value , receiver ) ;
if ( success )  {
                    log ( ’ property ’+ name + ’ on ’+ target + ’ set to ’+ value ) ;
                    }
                return success ;
```

```
                                    }
} ) ; // end of proxy

}
```

The Reflect.set method forwards the intercepted property assignment operation to the target object, returning whether or not the property was updated successfully.

## Proxy Traps

### The **get()** trap

The get() trap is fired when a property of the target object is accessed via the proxy object.

In the previous example, a message is printed out when a property of the user object is accessed by the proxyUser object.

Generally, you can develop a custom logic in the get() trap when a property is accessed.

For example, you can use the get() trap to define computed properties for the target object. The computed properties are properties whose values are calculated based on values of existing properties.

The user object does not have a property fullName, you can use the get() trap to create the fullName property based on the firstName and lastName properties:

```
const user = {
   firstName: 'John',
   lastName: 'Doe'
}

const handler = {
   get(target, property) {
      return property === 'fullName' ?
         `${target.firstName} ${target.lastName}` :
         target[property];
   }
};

const proxyUser = new Proxy(user, handler);
console.log(proxyUser.fullName);
```

## The set() trap

The set() trap controls behavior when a property of the target object is set.
Suppose that the age of user must be greater than 18. To enforce this constraint, you develop a set() trap as follows:

```
const user = {
    firstName: 'John',
    lastName: 'Doe',
    age: 20
}

const handler = {
    set(target, property, value) {
        if (property === 'age') {
            if (typeof value !== 'number') {
                throw new Error('Age must be a number.');
            }
            if (value < 18) {
                throw new Error('The user must be 18 or older.')
            }
        }
        target[property] = value;
    }
};

const proxyUser = new Proxy(user, handler);
```

First, set the age of user to a string:

```
proxyUser.age = 'foo';
```

Output:

```
Error: Age must be a number.
```

Second, set the age of the user to 16:

```
proxyUser.age = '16';
```

Output:

```
The user must be 18 or older.
```

Third, set the age of the user to 21:

```
proxyUser.age = 21;
```

## JavaScript Map object

Prior to ES6, when you need to map keys to values, you often use an object, because an object allows you to map a key to a value of any type.

However, using an object as a map has some side effects:
An object always has a default key like the prototype.

A key of an object must be a string or a symbol, you cannot use an object as a key.
An object does not have a property that represents the size of the map.
ES6 provides a new collection type called Map that addresses these deficiencies.

By definition, a Map object holds key-value pairs where values of any type can be used as either keys or values. In addition, a Map object remembers the original insertion order of the keys.
Example:
let map = new Map([iterable]);

## Initialize a map with an iterable object

```
let userRoles = new Map([
    [john, 'admin'],
    [lily, 'editor'],
    [peter, 'subscriber']
]);

let john = {name: 'John Doe'},
    lily = {name: 'Lily Bush'},
    peter = {name: 'Peter Drucker'};

let userRoles = new Map();
console.log(typeof(userRoles)); // object
console.log(userRoles instanceof Map); // true
```

Add elements to a Map

```
userRoles.set(john, 'admin');
userRoles.set(lily, 'editor') .set(peter, 'subscriber');
```

## Check the existence of an element by key

To check if a key exists in the map, you use the has() method.

```
userRoles.has(foo); // false
userRoles.has(lily); // true
```

## Get the number of elements in the map

The size property returns the number of entries in the map.

```
console.log(userRoles.size); // 3
```

## Iterate over map values and keys

```
for (let user of userRoles.keys()) {
    console.log(user.name);
}
// John Doe
// Lily Bush
// Peter Drucker
for (let role of userRoles.values()) {
    console.log(role);
}
// admin
// editor
// subscriber
```

## Convert map keys or values to a array

```
var keys = [...userRoles.keys()];
console.log(keys);
```

## Delete an element by key

```
userRoles.delete(john);
```

## Useful JavaScript Map() methods

- clear() – removes all elements from the map object.
- delete(key) – removes an element specified by the key. It returns if the element is in the map, or false if it does not.
- entries() – returns a new Iterator object that contains an array of [key, value] for each element in the map object. The order of objects in the map is the same as the insertion order.
- forEach(callback[, thisArg]) – invokes a callback for each key-value pair in the map in the insertion order. The optional thisArg parameter sets the this value for each callback.
- get(key) – returns the value associated with the key. If the key does not exist, it returns undefined.
- has(key) – returns true if a value associated with the key exists, otherwise, return false.
- keys() – returns a new Iterator that contains the keys for elements in insertion order.
- set(key, value) – sets the value for the key in the map object. It returns the map object itself therefore you can chain this method with other methods.
- values() returns a new iterator object that contains values for each element in insertion order

## WeakMap

A WeakMap is similar to a Map except the keys of a WeakMap must be objects. It means that when a reference to a key (an object) is out of scope, the corresponding value is automatically released from the memory.

A WeakMap only has subset methods of a Map object:

- get(key)
- set(key, value)
- has(key)
- delete(key)

Here are the main difference between a Map and a WeekMap:

- Elements of a WeakMap cannot be iterated.
- Cannot clear all elements at once.
- Cannot check the size of a WeakMap.

## JavaScript Set Type in ES6

Set allows you to manage a collection of unique values of any type effectively.

All elements in the set must be unique. Set that stores a collection of unique values of any type.

```
let setObject = new Set();
let setObject = new Set(iterableObject);
```

## Useful Set methods

The Set object provides the following useful methods:

- add(value) – appends a new element with a specified value to the set. It returns the Set object, therefore, you can chain this method with another Set method.
- clear()  – removes all elements from the Set object.
- delete(value) – deletes an element specified by the value.
- entries()– returns a new Iterator that contains an array of  [value, value] .
- forEach(callback [, thisArg]) – invokes a callback on each element of the Set with the this value sets to thisArg in each call.
- has(value) – returns true if an element with a given value is in the set, or false if it is not.
- keys() – is the same as values() function.
- [@@iterator] – returns a new Iterator object that contains values of all elements stored in the insertion order.

## JavaScript Set examples

## Create a new Set from an Array

```
let chars = new Set(['a', 'a', 'b', 'c', 'c']);
```

All elements in the set must be unique therefore the chars only contains 3 distinct elements a, b and c

```
console.log(chars);
Output:
Set { 'a', 'b', 'c' }
```

When you use the typeof operator to the chars, it returns object
.

```
console.log(typeof(chars));
Output:
Object
```

The chars set is an instance of the Set type so the following statement returns true.

```
let result = chars instanceof Set;
console.log(result); //true
```

## Get the size of a Set

```
let size = chars.size;
console.log(size);//  3
```

## add elem to set

```
chars.add('d');
console.log(chars);
```

examples:

1)

```
let exist = chars.has('a');
console.log(exist);// true
```

2)

```
chars.delete('f');
console.log(chars); // Set {"a", "b", "c", "d", "e"}
```

3)

```
chars.clear();
console.log(chars); // Set{}
```

4)

```
let roles = new Set();
roles.add('admin')
   .add('editor')
   .add('subscriber');

for (let role of roles) {
   console.log(role);
}
```

## WeakSets

- A WeakSet is similar to a Set except that it contains only objects.
- Since objects in a WeakSet may be automatically garbage-collected, a WeakSet does not have size property.
- Like a WeakMap, you cannot iterate elements of a WeakSet, therefore, you will find that WeakSet is rarely used in practice.
- In fact, you only use a WeakSet to check if a specified value is in the set. Here is an example:

```
let computer = {type: 'laptop'};
let server = {type: 'server'};
let equipment = new WeakSet([computer, server]);

if (equipment.has(server)) {
   console.log('We have a server');
}
```

Output

```
We have a server
```

## JAVASCRIPT

Javascript is a scripting language whose language runtime is often embedded within a larger execution environment. By far the most common execution environment for Javascript is the web browser.

While the full Javascript language as we know it today has a lot of accidental complexity as a side-effect of a complex evolutionary process, at its core it is a fairly simple dynamic language with first-class lexical closures and a concise object literal notation that makes it easy to create one-off anonymous objects.

In Javascript, objects are records of properties mapping names (strings) to values.
A simple twodimensional diagonal point can be defined as:

```
var point = {
                x : 5 ,
                get y ( ) { return this . x ; } ,
                toString : function ( ) { return ’ ( ’ + x + ’ , ’ + y + ’ ) ’ ] ;
}
```

ECMAScript 5 distinguishes between two kinds of properties.
Here, x is a data property, mapping a name to a value directly.

y is an accessor property, mapping a name to a “getter” and/or a “setter” function.
The expression point.y implicitly calls the getter function.

ECMAScript 5 further associates with each property a set of attributes. Attributes are meta-data that describe whether the property is writable (can be assigned to), enumerable (whether it appears in for-in loops) or configurable (whether the property can be deleted and whether its attributes can be modified1 ).

A non-configurable, no writable data property is in essence a constant binding. The following code snippet shows how these attributes can be inspected and defined:

```
var pd = Object . getOwnPropertyDescriptor ( point , ’ x ’ ) ;
/ / pd = {
 / / value : 5 ,
 / / w r i t a b l e : t r u e ,
/ / enumerable : t r u e ,
 / / c o nf i g u r a b l e : t r u e
/ / }

Object . defineProperty ( point , ’ z ’ , {
                    get : function ( ) { return this . x ; } ,
                    enumerable : false ,
                    configurable : true
                     } ) ;
```

The pd object and the third argument to defineProperty are called property descriptors. These are objects that describe properties of objects. Data property descriptors declare a value and a writable property, while accessor property descriptors declare a get and/or a set property. The Object.create function can be used to generate new objects based on a set of property descriptors directly. Its first argument specifies the prototype of the object to be created (Javascript uses object-based inheritance, further discussed in Section 4.3). Its second argument is an object mapping property names to property descriptors. We could have also defined the point object explicitly as:

```
var point = Object . create ( Object . prototype , {
                    x : { value : 5 , enumerable : true , writable : true ,
               configurable : true } ,
                    y : { get : function ( ) { return this . x ; } , enumerable : true , .
               . . } ,
                    toString : { value : function ( ) { . . . } , enumerable : true , . . .
               }
} ) ;
```

ECMAScript 5 supports the creation of tamper-proof objects that can protect themselves from modifications by client objects.
Objects can be made non-extensible, sealed or frozen.
By default, Javascript objects are extensible collections of properties.
However, a non-extensible object cannot be extended with new properties.
A sealed object is a non-extensible object whose own (non-inherited) properties

are all non-configurable. Finally, a frozen object is a sealed object
whose own data properties are all nonwritable.
The call Object.freeze(obj) freezes the object obj, effectively making the structure of obj (but not obj's property values) immutable.

## JavaScript FAQS:

## What is JavaScript?

**JavaScript** is *a scripting language*. It is different from Java language.
It is object-based, lightweight, cross-platform translated language.
It is widely used for client-side validation.
The JavaScript Translator (embedded in the browser) is responsible for translating the JavaScript code for the web browser.

### 1. List some features of JavaScript.

- Lightweight
- Interpreted programming language
- Good for the applications which are network-centric
- Complementary to Java
- Complementary to HTML
- Open source
- Cross-platform

### 2. List some of the advantages of JavaScript.

Some of the advantages of JavaScript are:

- Server interaction is less
- Feedback to the visitors is immediate
- Interactivity is high
- Interfaces are richer

### 3. List some of the disadvantages of JavaScript.

Some of the disadvantages of JavaScript are:

- No support for multithreading
- No support for multiprocessing
- Reading and writing of files is not allowed
- No support for networking applications.

4. Define a named function in JavaScript.

The function which has named at the time of definition is called a named function. For example

```
function msg()
{
  document.writeln("Named Function");
}
msg();
```

5. Name the types of functions

The types of function are:

Named - These type of functions contains name at the time of definition. For Example:

```
function display()
{
  document.writeln("Named Function");
}
display();
```

Anonymous - These type of functions doesn't contain any name. They are declared dynamically at runtime.

```
var display=function()
{
  document.writeln("Anonymous Function");
}
display();
```

6. Can an anonymous function be assigned to a variable?

Yes, you can assign an anonymous function to a variable.

7. Define closure.

In JavaScript, we need closures when a variable which is defined outside the scope in reference is accessed from some inner scope.

```
var num = 10;
function sum()
{
document.writeln(num+num);
}
sum();
```

8. If we want to return the character from a specific index which method is used?

The JavaScript string charAt() method is used to find out a char value present at the specified index. The index number starts from 0 and goes to n-1, where n is the length of the string. The index value can't be a negative, greater than or equal to the length of the string. For example:

```
var str="Javatpoint";
document.writeln(str.charAt(4));
```

## 9. What is the difference between JavaScript and JScript?

Netscape provided the JavaScript language. Microsoft changed the name and called it JScript to avoid the trademark issue. In other words, you can say JScript is the same as JavaScript, but Microsoft provides it.

## 10. How to write a hello world example of JavaScript?

A simple example of JavaScript hello world is given below. You need to place it inside the body tag of HTML.

```
<script type="text/javascript">
document.write("JavaScript Hello World!");
</script>
```

## 11. How to use external JavaScript file?

I am assuming that js file name is message.js, place the following script tag inside the head tag.

```
<script type="text/javascript" src="message.js"></script>
```

## 12. What is BOM?

BOM stands for Browser Object Model. It provides interaction with the browser. The default object of a browser is a window. So, you can call all the functions of the window by specifying the window or directly.

## 13. What is DOM? What is the use of document object?

DOM stands for Document Object Model. A document object represents the HTML document. It can be used to access and change the content of HTML.

## 14. What is the use of window object?

The window object is created automatically by the browser that represents a window of a browser. It is not an object of JavaScript. It is a browser object.

The window object is used to display the popup dialog box. Let's see with description.

| **Method** | **Description** |
|---|---|
| alert() | displays the alert box containing the message with ok button. |
| confirm() | displays the confirm dialog box containing the message with ok and cancel button. |
| prompt() | displays a dialog box to get input from the user. |
| open() | opens the new window. |

| | |
|---|---|
| close() | closes the current window. |
| setTimeout() | performs the action after specified time like calling function, evaluating expressions. |

## 15. What is the use of history object?

The history object of a browser can be used to switch to history pages such as back and forward from the current page or another page. There are three methods of history object.

history.back() - It loads the previous page.

history.forward() - It loads the next page.

history.go(number) - The number may be positive for forward, negative for backward. It loads the given page number.

## 16. What are the JavaScript data types?

There are two types of data types in JavaScript:

1. Primitive Data Types - The primitive data types are as follows:

| Data Type | Description |
|---|---|
| | uence of characters, e.g., "hello" |
| | ric values, e.g., 100 |
| | an value either false or true |
| | defined value |
| | e., no value at all |

2. Non-primitive Data Types - The non-primitive data types are as follows:

| Data Type | Description |
|---|---|
| | tance through which we can access members |
| | ıp of similar values |
| | ır expression |

### 17. What is the difference between == and ===?

The == operator checks equality only whereas === checks equality, and data type, i.e., a value must be of the same type.

### 18. How to write HTML code dynamically using JavaScript?

The innerHTML property is used to write the HTML code using JavaScript dynamically. Let's see a simple example:

```
document.getElementById('mylocation').innerHTML="<h2>This is heading using
                        JavaScript</h2>";
```

### 19. How to write normal text code using JavaScript dynamically?

The innerText property is used to write the simple text using JavaScript dynamically. Let's see a simple example:

```
document.getElementById('mylocation').innerText="This is text using JavaScript";
```

### 20. Difference between Client side JavaScript and Server side JavaScript?

Client-side JavaScript comprises the basic language and predefined objects which are relevant to running JavaScript in a browser.
The client-side JavaScript is embedded directly by in the HTML pages.
The browser interprets this script at runtime.

Server-side JavaScript also resembles client-side JavaScript.
It has a relevant JavaScript which is to run in a server.
The server-side JavaScript are deployed only after compilation.

### 21. How to set the cursor to wait in JavaScript?

The cursor can be set to wait in JavaScript by using the property "cursor". The following example illustrates the usage:

```
<script>
window.document.body.style.cursor = "wait";
</script>
```

### 22. What is this [[[]]]?

This is a three-dimensional array.

```
var myArray = [[[]]];
```

23. What are the pop-up boxes available in JavaScript?

Alert Box
Confirm Box
Prompt Box

**Example of alert() in JavaScript**

```
<script type="text/javascript">
function msg(){
 alert("Hello Alert Box");
}
</script>
<input type="button" value="click" onclick="msg()"/>
```

**Example of confirm() in JavaScript**

```
<script type="text/javascript">
function msg(){
var v= confirm("Are u sure?");
if(v==true){
alert("ok");
}
else{
alert("cancel");
}

}
</script>

<input type="button" value="delete record" onclick="msg()"/>
```

Example of prompt() in JavaScript

```
<script type="text/javascript">
function msg(){
var v= prompt("Who are you?");
alert("I am "+v);

}
</script>
<input type="button" value="click" onclick="msg()"/>
```

24. How can we detect OS of the client machine using JavaScript?

The navigator.appVersion string can be used to detect the operating system on the client machine.

25. How to submit a form using JavaScript by clicking a link?

Let's see the JavaScript code to submit the form by clicking the link.

```
<form name="myform" action="index.php">
```

```
Search: <input type='text' name='query' />

<a href="javascript: submitform()">Search</a>
</form>

<script type="text/javascript">
function submitform()
{
  document.myform.submit();
}
</script>
```

26. How to change the background color of HTML document using JavaScript?

```
<script type="text/javascript">
document.body.bgColor="pink";
</script>
```

27. How to validate a form in JavaScript?

```
<script>
function validateform(){
var name=document.myform.name.value;
var password=document.myform.password.value;

if (name==null || name==""){
  alert("Name can't be blank");
  return false;
}else if(password.length<6){
  alert("Password must be at least 6 characters long.");
  return false;
  }
}

</script>
<body>
<form name="myform" method="post" action="abc.jsp" onsubmit="return validateform()"
                                 >
Name: <input type="text" name="name"><br/>
Password: <input type="password" name="password"><br/>
<input type="submit" value="register">
</form>
```

28. How to validate email in JavaScript?

```
<script>
function validateemail()
{
var x=document.myform.email.value;
var atposition=x.indexOf("@");
var dotposition=x.lastIndexOf(".");
if (atposition<1 || dotposition<atposition+2 || dotposition+2>=x.length){
  alert("Please enter a valid e-
mail address \n atpostion:"+atposition+"\n dotposition:"+dotposition);
  return false;
  }
}
</script>
<body>
<form name="myform"  method="post" action="#" onsubmit="return validateemail();">
Email: <input type="text" name="email"><br/>

<input type="submit" value="register">
</form>
```

## 29. What is the use of debugger keyword in JavaScript?

JavaScript debugger keyword sets the breakpoint through the code itself. The debugger stops the execution of the program at the position it is applied. Now, we can start the flow of execution manually. If an exception occurs, the execution will stop again on that particular line.. For example:

```
function display()
{
x = 10;
y = 15;
z = x + y;
debugger;
document.write(z);
document.write(a);
}
display();
```

## 30. What is the use of a WeakMap object in JavaScript?

The JavaScript WeakMap object is a type of collection which is almost similar to Map. It stores each element as a key-value pair where keys are weakly referenced. Here, the keys are objects and the values are arbitrary values. For example:

```
function display()
{
var wm = new WeakMap();
var obj1 = {};
var obj2 = {};
var obj3= {};
wm.set(obj1, "jQuery");
wm.set(obj2, "AngularJS");
wm.set(obj3,"Bootstrap");
document.writeln(wm.has(obj2));
}
display();
```

## 31. What are the new features introduced in ES6?

The new features that are introduced in ES6 are listed as follows:

- Let and const keywords.
- Default Parameters.
- Arrow functions.
- Template Literals.
- Object Literals.
- Rest and spread operators.
- Destructuring assignment.
- Modules, Classes, Generators, and iterators.
- Promises, and many more.

## 32. Define let and const keywords.

**let:** The variables declared using **let** keyword will be mutable, i.e.,
the values of the variable can be changed. It is similar to **var** keyword except that it provides block scoping.
**const:** The variables declared using the const keyword are immutable and block-scoped.
The value of the variables cannot be changed or re-assigned if they are declared by using the const keyword.

## 33. What is the arrow function, and how to create it?

Arrow functions are introduced in ES6. Arrow functions are the shorthand notation to write ES6 functions. The definition of the arrow function consists of parameters, followed by an arrow (=>) and the body of the function.
An Arrow function is also called as 'fat arrow' function.
We cannot use them as constructors.
Syntax

```
const functionName = (arg1, arg2, ...) => {
   //body of the function
}
```

## 34. Give an example of an Arrow function in ES6? List down its advantages.

Arrow function provides us a more accurate way of writing the functions in JavaScript. They allow us to write smaller function syntax.

The context within the arrow functions is lexically or statically scoped. Arrow functions do not include any prototype property, and cannot be used with the new keyword. You can learn more about arrow functions by clicking on this link ES6 Arrow Function.

**Example**

```
var myfun = () => {
    console.log("It is an Arrow Function");
   };
   myfun();
```

**Output**

```
It is an Arrow Function
```

### Advantages of Arrow Function

The advantages of the arrow function are listed below:

- It reduces code size.
- The return statement is optional for a single line function.
- Lexically bind the context.
- Functional braces are optional for a single-line statement.

## 35. Discuss spread operator in ES6 with an example.

The spread operator is represented by three dots (...) to obtain the list of parameters. It allows the expansion of an iterable such as array or string in places where more than zero arguments are expected.
The spread operator syntax is similar to the rest operator, but functionality is entirely opposite to it. It is also used to combine or to perform the concatenation between arrays. Let's understand it by an example.
**Example**

```
let num1 = [40,50,60];

let num2 = [10,20,30,...num1,70,80,90,100];

console.log(num2);
```

**Output**

```
[
  10, 20, 30, 40,  50,
  60, 70, 80, 90, 100
]
```

36. Discuss the Rest parameter in ES6 with an example.
It is introduced in ES6 that improves the ability to handle the parameters.
With rest parameters, it is possible to represent indefinite parameters as an array.
By using the rest parameter, we can call a function with any number of arguments.

**Example**

```
function show(...args) {
   let sum = 0;
   for (let i of args) {
      sum += i;
   }
   console.log("Sum = "+sum);
 }

 show(10, 20, 30);
```

37. What are the template literals in ES6?
Template literals are a new feature introduced in ES6.
It provides an easy way of creating multiline strings and perform string interpolation.
Template literals allow embedded expressions and also called as string literals.
Prior to ES6, template literals were referred to as **template strings**.
Template literals are enclosed by the **backtick (` `) character**.
Placeholders in template literals are represented by the dollar sign and
the curly braces **(${expression})**. If we require to use an expression
within the backticks, then we can place that expression in the **(${expression})**.
To learn more about template literals in ES6, follow this link ES6 Template Literals.

**Example**

```
let str1 = "Hello";

let str2 = "World";

let str = `${str1} ${str2}`;
console.log(str);
```

38. Discuss Destructuring Assignment in ES6.
Destructuring is introduced in ECMAScript 2015 or ES6 to extract data from objects
and arrays into separate variables. It allows us to extract smaller fragments
from objects and arrays.
To learn more about array destructuring in ES6, follow this link ES6 Array Destructuring.

To learn more about object destructuring in ES6, follow this link
ES6 Object Destructuring.
**Example**

```
let fullname =['Alan','Rickman'];
let [fname,lname] = fullname;
console.log (fname,lname);
```

**Output**

```
Alan Rickman
```

39. Define set.
A set is a data structure that allows us to create a collection of unique values. It is a collection of values that are similar to arrays, but it does not include any duplicates. It supports both object references and primitive values.

To learn more about Sets in ES6, follow this link ES6 Sets.

**Example**

let colors = new Set(['Green', 'Red', 'Orange', 'Yellow', 'Red']);
console.log(colors);
Output
Set { 'Green', 'Red', 'Orange', 'Yellow' }

40. Define Map.
Prior to ES6, when we require the mapping of keys and values, we often use an object. **Map object** is a new collection type, which is introduced in ES6. It holds the key-value pairs in which any type of values can be used as either keys or values.
A map object always remembers the actual insertion order of the keys. Maps are ordered, so they traverse the elements in their insertion order.

41. What do you understand by Weakset?
Using weakset, it is possible to store weakly held objects in a collection. As similar to set, weakset cannot store duplicate values. Weakset cannot be iterated.
Weakset only includes add(value), delete(value) and has(value) methods of the set object.

42. What do you understand by Weakmap?

Weak maps are almost similar to maps, but the keys in weak maps must be objects. It stores each element as a key-value pair where keys are weakly referenced. Here, the keys are objects, and the values are arbitrary.
A weak map object iterates the element in their insertion order. It only includes **delete(key), get(key), has(key)** and **set(key, value)** method.

43. What do you understand by Callback and Callback hell in JavaScript?

- **Callback:** It is used to handle the execution of function after the completion of the execution of another function. A callback would be helpful in working with events. In the callback, a function can be passed as an argument to another function. It is a great way when we are dealing with basic cases such as minimal asynchronous operations.

- **Callback hell:** When we develop a web application that includes a lot of code, then working with callback is messy. This excessive Callback nesting is often referred to as **Callback hell**.

44. What do you understand by the term Hoisting in JavaScript?

It is a JavaScript's default behavior, which is used to move all the declarations at the top of the scope before the execution of code. It can be applied to functions as well as on variables. It allows the JavaScript to use the component before its declaration. It does not apply to scripts that run in strict mode.

45. Define Babel.

Babel is one of the popular transpilers of JavaScript. It is mainly used for converting the ES6 plus code into the backward-compatible version of JavaScript that can be run by previous JavaScript engines.

46. Following are the features of JavaScript:

- It is a lightweight, interpreted programming language.
- It is designed for creating network-centric applications.
- It is complementary to and integrated with Java.
- It is an open and cross-platform scripting language.

47. Following are the advantages of using JavaScript −

- **Less server interaction** − You can validate user input before sending the page off to the server. This saves server traffic, which means less load on your server.
- **Immediate feedback to the visitors** − They don't have to wait for a page reload to see if they have forgotten to enter something.
- **Increased interactivity** − You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard.
- **Richer interfaces** − You can use JavaScript to include such items as drag-and-drop components and sliders to give a Rich Interface to your site visitors.

48. Remove duplicates items from an array using ES6 Set?

- The Set object lets you store unique values of any type, whether primitive values or object
- Here's a one-liner to remove duplicates from an array. (ES6, of course!)

```
const numbers = [1, 2, 3, 4, 5, 5, 5, 5, 5, 5];
function removeDuplicates(array) {
return [...new Set(array)];
}
console.log(removeDuplicates(numbers)); // [1, 2, 3, 4, 5]
```

## 49. Map & Set:

**Map:**

Map is a collection of keyed data items, just like an Object.
Note - Map allows keys of any type.

- `new Map()` – creates the map.
- `map.set(key, value)` – stores the value by the key.
- `map.get(key)` – returns the value by the key, `undefined` if `key` doesn't exist in map.
- `map.has(key)` – returns `true` if the `key` exists, `false` otherwise.
- `map.delete(key)` – removes the value by the key.
- `map.clear()` – removes everything from the map.
- `map.size` – returns the current element count.

```
map = new Map()
map.set("website", "Full Stack Tutorials")
map.set("topic", "JavaScript ES6");

console.log(map);

//Output
/*
Map {
  'website' => 'Full Stack Tutorials',
  'topic' => 'JavaScript ES6'
}
*/
```

**Set:**

Set is a collection of unique data items, without keys.

- `new Set(iterable)` – creates the set, and if an `iterable` object is provided (usually an array), copies values from it into the set.
- `set.add(value)` – adds a value, returns the set itself.
- `set.delete(value)` – removes the value, returns `true` if `value` existed at the moment of the call, otherwise `false`.
- `set.has(value)` – returns `true` if the value exists in the set, otherwise `false`.
- `set.clear()` – removes everything from the set.
- `set.size` – is the elements count.

## 50. Difference between Arrow Function and Normal Function in JavaScript?

There are following differences between Arrow Function and Normal (Regular) Function:

**1. Syntax:**

Both Arrow Function and Normal (Regular) Function has different syntax.

```
//Normal Function: let func = function myFunc(params){ // body of the function }; OR function
myFunc(params){ // body of the function };
//Arrow Function: let func = (params) => { // body of the function };
```

**2. Use of this keyword:**

Unlike regular functions, arrow functions do not have their own this.

```
let myFunc = { name: "Full Stack Tutorials", regFunc() { console.log(`Welcome to, ${this.name}`); //
'this' binding works }, arrowFunc: () => { console.log(`Welcome to, ${this.name}`); // no 'this' binding }
}; myFunc.regFunc(); myFunc.arrowFunc();
```

**3. Using new keyword:**

Arrow functions cannot be used as constructor with new, it will throw error.

```
let myFunc = () => {}; let func = new myFunc(); // Uncaught TypeError: myFunc is not a constructor
```

**4. Availability of arguments objects:**

Arrow functions don't have their own arguments object. Therefore, arguments is simply a reference to the arguments of the enclosing scope.

**5. Use of prototype property:**

Arrow functions do not have a prototype property.

```
let myFunc = () => {}; console.log(myFunc.prototype); // undefined
```

## 51. What is the use of isNaN function ?

The Number.isNan function in JavaScript is used to determines whether the passed value is NaN (Not a Number) and is of the type “Number”. In JavaScript, the value NaN is considered a type of number. It returns true if the argument is not a number else it returns false.

## 52. Write a JavaScript code for adding new elements dynamically ?

```
<!DOCTYPE html>
<html>

<head>
    <title>
        JavaScript code for adding new
        elements dynamically
    </title>
</head>

<body>
    <button onclick="create()">
        Click Here!
    </button>

    <script>
        function create() {
            var geeks = document.createElement('geeks');
            geeks.textContent = "Geeksforgeeks";
            geeks.setAttribute('class', 'note');
            document.body.appendChild(geeks);
        }
    </script>
</body>
```

53. What are global variables? How are these variable declared and what are the problems associated with them ?

In contrast, global variables are the variables that are defined outside of functions. These variables have a global scope, so they can be used by any function without passing them to the function as parameters.

**Example:**

```
<script>
     var petName = "Rocky"; //Global Variable
     myFunction();

     function myFunction() {
         document.getElementById("geeks").innerHTML
                      = typeof petName + "- " +
                      "My pet name is " + petName;
     }

     document.getElementById("Geeks").innerHTML
                      = typeof petName + "- " +
                      "My pet name is " + petName;
</script>
```

It is difficult to debug and test the code that relies on global variables

The End