### 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 and callback 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.

The 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.

In ES5, 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](https://www.javascripttutorial.net/es-next/javascript-globalthis/) as [properties](https://www.javascripttutorial.net/javascript-object-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](http://www.javascripttutorial.net/es6/javascript-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](http://www.javascripttutorial.net/javascript-array/), a  [map](https://www.javascripttutorial.net/es6/javascript-map/), or a [set](https://www.javascripttutorial.net/es6/javascript-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](http://www.javascripttutorial.net/javascript-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](https://www.javascripttutorial.net/javascript-objects/) literal is one of the most popular [patterns for creating objects in JavaScript](https://www.javascripttutorial.net/create-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](https://www.javascripttutorial.net/javascript-array/), [String](https://www.javascripttutorial.net/javascript-string/), [Map](https://www.javascripttutorial.net/es6/javascript-map/), [Set](https://www.javascripttutorial.net/es6/javascript-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}`);

}

1. **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](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) to assign the score property of the current iterated element to the score variable.

1. **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);

}

1. **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);

}

1. for...of**vs.**for...in

* The for...in iterates over all [enumerable properties](https://www.javascripttutorial.net/javascript-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](http://www.javascripttutorial.net/javascript-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](http://www.javascripttutorial.net/javascript-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](http://www.javascripttutorial.net/javascript-function-type/) 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](https://www.javascripttutorial.net/es6/javascript-object-destructuring/) or elements of an [array](http://www.javascripttutorial.net/javascript-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](https://www.javascripttutorial.net/javascript-object-properties/) of an [object](https://www.javascripttutorial.net/javascript-objects/) 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() {

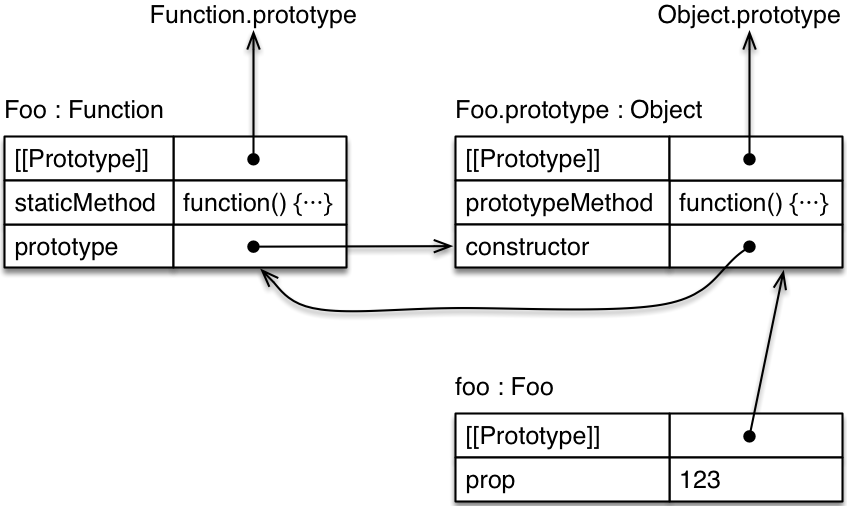
**return** 'prototypical';

}

}

**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*

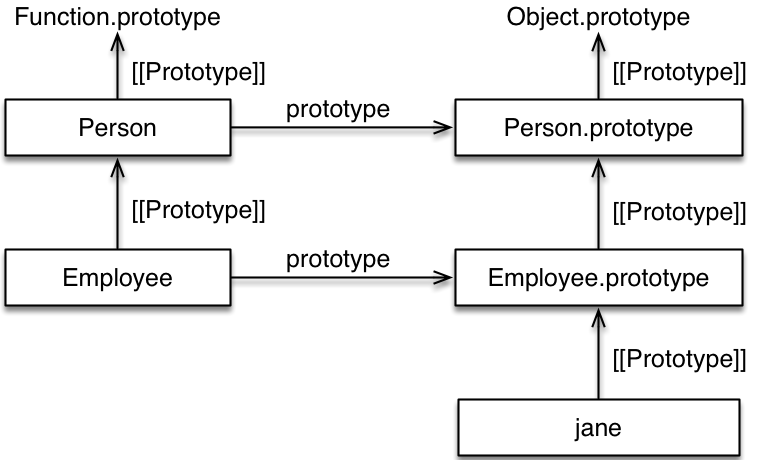
**new** D().logProp();

*// 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)*

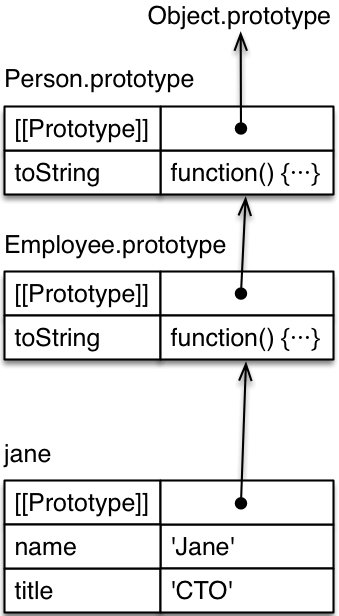
}

}

**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

Symbols are a new primitive type in ECMAScript 6. 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)

### 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

### 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](http://speakingjs.com/es5/ch18.html#_pitfall_array_like_objects), which have a property length and indexed elements. Examples include the results of DOM operations such as document.getElementsByClassName().
* [Iterable values](https://exploringjs.com/es6/ch_iteration.html#ch_iteration), 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'] |
| copyWithinES6 | Preserved | [,'a','b',,].copyWithin(2,0) → [,'a',,'a'] |
| entriesES6 | Elements | [...[,'a'].entries()] → [[0,undefined], [1,'a']] |
| every | Ignored | [,'a'].every(x => x==='a') → true |
| fillES6 | Filled | new Array(3).fill('a') → ['a','a','a'] |
| filter | Removed | ['a',,'b'].filter(x => true) → ['a','b'] |
| findES6 | Elements | [,'a'].find(x => true) → undefined |
| findIndexES6 | 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##' |
| keysES6 | 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 |
| valuesES6 | 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](http://www.javascripttutorial.net/javascript-function/)or constructor was called using the new operator.

The new.target is available for all functions. However, in [arrow functions](http://www.javascripttutorial.net/es6/javascript-arrow-function/), 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](http://www.javascripttutorial.net/es6/javascript-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

### 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.

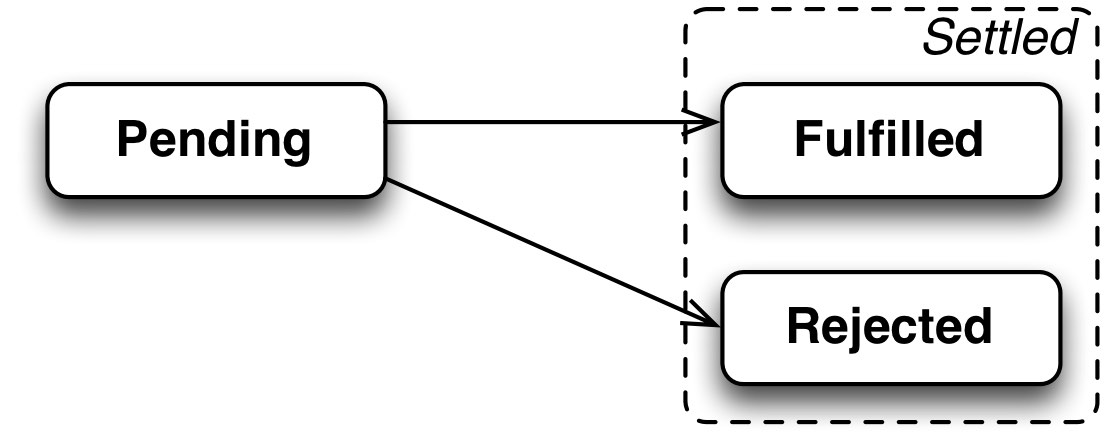
* ES6 modules are automatically strict-mode code, even if you don’t write "use strict"; in them.
* You can use import and export in modules.

### 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)*

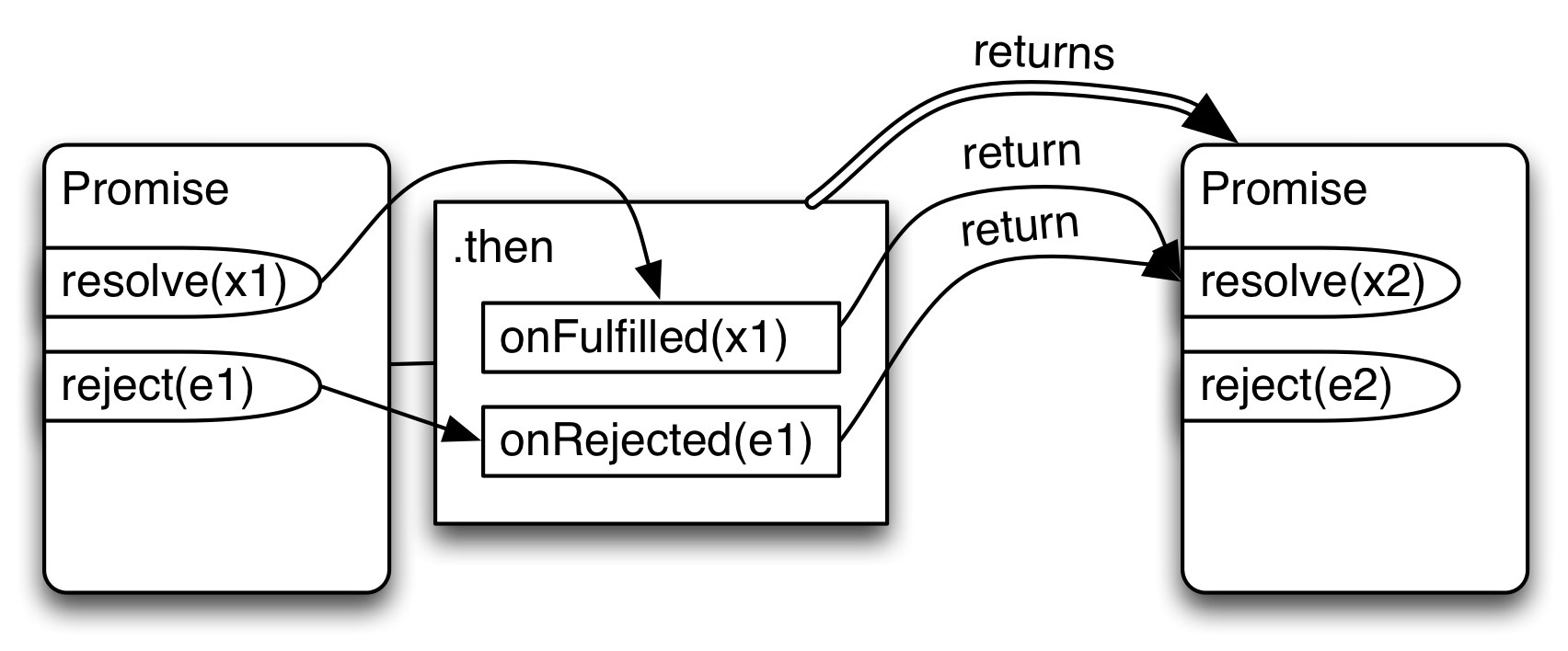
};

}

···

**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');

}

Note the new syntax: function\* is a new “keyword” for generator functions (there are also generator methods). yield is an operator with which a generator can pause itself. Additionally, generators can also receive input and send output via yield.

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();

1. Generator function expressions:

**const** genFunc = **function**\* () { ··· };

**const** genObj = genFunc();

1. Generator method definitions in object literals:

**const** obj = {

\* generatorMethod() {

···

}

};

**const** genObj = obj.generatorMethod();

1. 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\*.

### **yield\*: the full story**

### 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.

### 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](https://exploringjs.com/deep-js/ch_proxies.html#forwarding-intercepted-operations), 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**

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 )

### Proxies

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 ) ;

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

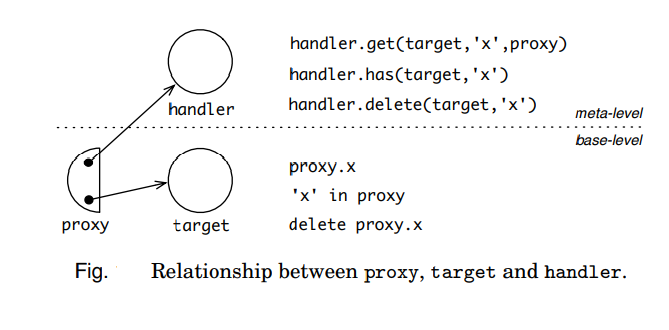
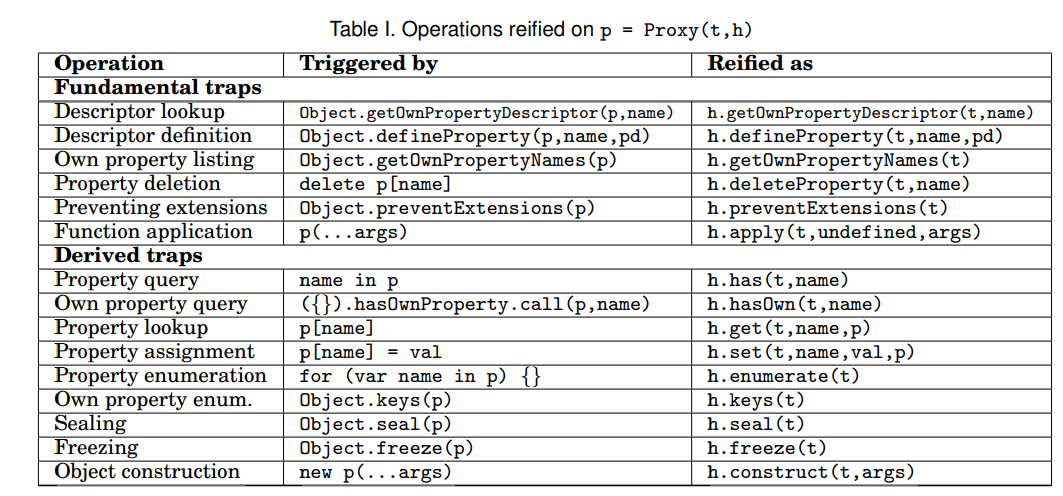


Table I lists those base-level operations applicable to objects that can be trapped by handlers. An intercepted operation mostly simply triggers the corresponding trap and returns its result. The enumerate trap must return an array of strings representing the enumerable property names of the proxy. The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



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.

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.