[**https://leanpub.com/understandinges6/read#leanpub-auto-block-level-declarations**](https://leanpub.com/understandinges6/read#leanpub-auto-block-level-declarations)

[**https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/for...of**](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Statements/for...of)

#### Difference between for...of and for...in

The [for...in](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...in) loop will iterate over all enumerable properties of an object.

The for...of syntax is specific to **collections**, rather than all objects. It will iterate in this manner over the elements of any collection that has a [Symbol.iterator] property.

**// For**

for (i = 0; i < 5; i++) {  
    text += "The number is " + i + "<br>";  
}

**// For/in**

var person = {fname:"John", lname:"Doe", age:25};   
  
var text = "";  
var x;  
for (x in person) {  
    text += person[x];  
}

**// For/of**

let iterable = new Map([['a', 1], ['b', 2], ['c', 3]]);

for (let entry of iterable) {

console.log(entry);

}

// ['a', 1]

// ['b', 2]

// ['c', 3]

for (let [key, value] of iterable) {

console.log(value);

}

// 1

// 2

// 3

#### immediately-invoked function expressions

var funcs = [];

for (var i = 0; i < 10; i++) {

funcs.push((function(value) {

return function() {

console.log(value);

}

}(i)));

}

funcs.forEach(function(func) {

func(); // outputs 0, then 1, then 2, up to 9

});

#### The Temporal Dead Zone

if (condition) {

console.log(typeof value); // ReferenceError!

let value = "blue";

}

#### capitalizeFirstLetter

String.prototype.capitalizeFirstLetter = function() {

return this.charAt(0).toUpperCase() + this.slice(1);

}

'string'.capitalizeFirstLetter() // String

#### Better Unicode Support (32 bit) codePointAt()

#### Template Literals

Multiline Strings the Easy Way

**let** **message** **=** **`Multiline**

**string`;**

console.log(message); // "Multiline

// string"

console.log(message.length); // 16

**let message = `Multiline\nstring`;**

#### Making Substitutions (${name})

// eg 1

let name = "Nicholas",

message = "Hello, ${name}."; // Used " code

console.log(message); // "Hello, Nicholas.

// Eg 2

name = "Nicholas",

message = `Hello, ${name}.`; // Used ` code

console.log(message); // "Hello, Nicholas."

// Eg 3

let count = 10,

price = 0.25,

message1 = `${count} items cost $${(count \* price).toFixed(2)}.`;

console.log(message1); // "10 items cost $2.50."

#### Rest Parameters

// Unnamed paramenters :: ...keys is rest parameter  
function pick(object, **...keys**) {  
 console.log("Object %o", object);   
 let result = Object.create(null);  
 // start at the second parameter  
 for (let i = 0, len = keys.length; i < len; i++) {  
 result[keys[i]] = object[keys[i]];  
 }  
 return result;  
}

let book = {  
 title: "Understanding ECMAScript 6",  
 author: "Nicholas C. Zakas",  
 year: 2015  
};

let bookData = pick(book, "author", "year");  
console.log(bookData.author); // "Nicholas C. Zakas"  
console.log(bookData.year); // 2015  
console.log("bookData %o", bookData);

1. Rest parameters do not affect a function’s length property, which indicates the number of named parameters for the function(**object**). The value of length for pick() in this example is 1 because only object counts towards this value. **...keys** is unnamed parameter.

#### Rest Parameter Restrictions

1. The first restriction is that there can be only one rest parameter, and the rest parameter must be last. For example, this code won’t work:

// Syntax error: Can't have a named parameter after rest parameters

function pick(object, ...keys, last) {

let result = Object.create(null);

for (let i = 0, len = keys.length; i < len; i++) {

result[keys[i]] = object[keys[i]];

}

return result;

}

1. The second restriction is that rest parameters cannot be used in an object literal setter. That means this code would also cause a syntax error:

let object = {

// Syntax error: Can't use rest param in setter

set name(...value) {

// do something

}

};

Shallow copy :: where object references are shared when property values are objects

Deep Copy

#### Concise Methods

ECMAScript 6 also improves the syntax for assigning methods to object literals. In ECMAScript 5 and earlier, you must specify a name and then the full function definition to add a method to an object, as follows:

var person = {

name: "Nicholas",

sayName: function() {

console.log(this.name);

}

};

In ECMAScript 6, the syntax is made more concise by eliminating the colon and the function keyword. That means you can rewrite the previous example like this:

var person = {

name: "Nicholas",

**sayName**() {

console.log(this.name);

}

};

**Summary**

Objects are the center of programming in JavaScript, and ECMAScript 6 made some helpful changes to objects that both make them easier to deal with and more powerful.

ECMAScript 6 makes several changes to object literals. Shorthand property definitions make assigning properties with the same names as in-scope variables easier. Computed property names allow you to specify non-literal values as property names, which you’ve already been able to do in other areas of the language. Shorthand methods let you type a lot fewer characters in order to define methods on object literals, by completely omitting the colon and function keyword. ECMAScript 6 loosens the strict mode check for duplicate object literal property names as well, meaning you can have two properties with the same name in a single object literal without throwing an error.

The Object.assign() method makes it easier to change multiple properties on a single object at once. This can be very useful if you use the mixin pattern. The Object.is() method performs strict equality on any value, effectively becoming a safer version of === when dealing with special JavaScript values.

Enumeration order for own properties is now clearly defined in ECMAScript 6. When enumerating properties, numeric keys always come first in ascending order followed by string keys in insertion order and symbol keys in insertion order.

It’s now possible to modify an object’s prototype after it’s already created, thanks to ECMAScript 6’s Object.setPrototypeOf() method.

Finally, you can use the super keyword to call methods on an object’s prototype. The this binding inside a method invoked using super is set up to automatically work with the current value of this.

**Object Destructuring**

Object destructuring syntax uses an object literal on the left side of an assignment operation. For example:

let options = {

repeat: true,

save: false

};

// extract data from the object

let repeat = options.repeat,

save = options.save;

let node = {

type: "Identifier",

name: "foo"

};

**let { type, name } = node;**

console.log(type); // "Identifier"

console.log(name); // "foo"

In this example, type and name are initialized with values when declared, and then two variables with the same names are initialized with different values. The next line uses destructuring assignment to change those values by reading from the node object. Note that you must put parentheses around a destructuring assignment statement. That’s because an opening curly brace is expected to a be a block statement, and a block statement cannot appear on the left side of an assignment. The parentheses signal that the next curly brace is not a block statement and should be interpreted as an expression, allowing the assignment to complete

A destructuring assignment expression evaluates to the right side of the expression (after the =). That means you can use a destructuring assignment expression anywhere a value is expected. For instance, passing a value to a function:

let node = {

type: "Identifier",

name: "foo"

},

type = "Literal",

name = 5;

// assign different values using destructuring

({ type, name } = node);

console.log(type); // "Identifier"

console.log(name); // "foo"

##### Assigning to Different Local Variable Names

let node = {

type: "Identifier",

name: "foo"

};

**let { type: localType, name: localName } = node;**

console.log(localType); // "Identifier"

console.log(localName); // "foo"

This code uses destructuring assignment to declare the localType and localName variables, which contain the values from the node.type and node.name properties, respectively. The syntax type: localType says to read the property named type and store its value in the localType variable. This syntax is effectively the opposite of traditional object literal syntax, where the name is on the left of the colon and the value is on the right. In this case, the name is on the right of the colon and the location of the value to read is on the left.

You can add default values when using a different variable name, as well. The equals sign and default value are still placed after the local variable name. For example:

let node = {

type: "Identifier"

};

let { type: localType, **name: localName = "bar"** } = node;

console.log(**localType**); // "Identifier"

console.log(localName); // "bar"

##### **Nested Object Destructuring**

let node = {

type: "Identifier",

name: "foo",

loc: {

start: {

line: 1,

column: 1

},

end: {

line: 1,

column: 4

}

}

};

let { loc: { start }} = node;

console.log(start.line); // 1

console.log(start.column); // 1

You can go one step further and use a different name for the local variable as well:

let node = {

type: "Identifier",

name: "foo",

loc: {

**start: {**

line: 1,

column: 1

},

end: {

line: 1,

column: 4

}

}

};

// extract node.loc.start

**let { loc: { start: localStart }} = node;**

console.log(**localStart.line**); // 1

console.log(localStart.column); // 1

**Array Destructuring**

let colors = [ "red", "green", "blue" ];

let [ firstColor, secondColor ] = colors;

console.log(firstColor); // "red"

console.log(secondColor); // "green"

======

let colors = [ "red", "green", "blue" ];

let [ , , thirdColor ] = colors;

console.log(**thirdColor**); // "blue"

##### Destructuring Assignment

let colors = [ "red", "green", "blue" ],

firstColor = "black",

secondColor = "purple";

[ firstColor, secondColor ] = colors;

console.log(firstColor); // "red"

console.log(secondColor); // "green"

**Swapping variables in ECMAScript 5**

let a = 1,

b = 2,

tmp;

**tmp = a;**

**a = b;**

**b = tmp;**

console.log(a); // 2

console.log(b); // 1

**Swapping variables in ECMAScript 6**

let a = 1,

b = 2;

**[ a, b ] = [ b, a ];**

console.log(a); // 2

console.log(b); // 1

##### **Nested Destructuring**

let colors = [ "red", [ "green", "lightgreen" ], "blue" ];

// later  
let [ firstColor, [ secondColor, thirdColor, fourthColor="darkGreen" ], fifthColor ] = colors;

console.log(firstColor); // "red"  
console.log(secondColor); // "green"  
console.log(thirdColor); // "green"  
console.log(fourthColor); // "green"  
console.log(fifthColor); // "green"

##### **Rest Items**

let colors = [ "red", "green", "blue" ];

let [ firstColor, ...restColors ] = colors;

console.log(firstColor); // "red"

console.log(restColors.length); // 2

console.log(restColors[0]); // "green"

console.log(restColors[1]); // "blue"

**cloning an array in ECMAScript 5**

var colors = [ "red", "green", "blue" ];

var clonedColors = colors.concat();

console.log(clonedColors); //"[red,green,blue]"

**// cloning an array in ECMAScript 6**

let colors = [ "red", "green", "blue" ];

let [ ...clonedColors ] = colors;

console.log(clonedColors); //"[red,green,blue]"

**Destructured Parameters**

// properties on options represent additional parameters

function setCookie(name, value, **options**) {

options = options || {};

let secure = options.secure,

path = options.path,

domain = options.domain,

expires = options.expires;

*// code to set the cookie*

}

// third argument maps to options

setCookie("type", "js", {

secure: true,

expires: 60000

});

============

function setCookie(name, value, **{ secure, path, domain, expires }**) {

*// code to set the cookie*

}

setCookie("type", "js", **{**

**secure: true,**

**expires: 60000**

**});**

#### Destructured Parameters are Required

// Error!

setCookie("type", "js");

The third argument is missing, and so it evaluates to undefined as expected. This causes an error because destructured parameters are really just a shorthand for destructured declaration. When the setCookie() function is called, the JavaScript engine actually does this:

function setCookie(name, value, options) {  
 let { secure, path, domain, expires } = options;  
 // code to set the cookie  
}

function setCookie(name, value, { secure, path, domain, expires } = {}) {

// ...

}

#### Default Values for Destructured Parameters

You can specify destructured default values for destructured parameters just as you would in destructured assignment. Just add the equals sign after the parameter and specify the default value. For example:

function setCookie(name, value,

{

**secure = false,**

path = "/",

domain = "example.com",

expires = new Date(Date.now() + 360000000)

} = {}

) {

// ...

}