Learning ECMAScript 6

This book will provide step-by-step instructions on how to use ES6 features instead of old JavaScript hacks and methods.

The book will start by introducing all the built-in objects of ES6 and how to create ES6 custom iterators. Next, it will teach you how to write asynchronous code in a synchronous style using ES6.

Moving on, the book will teach how to use Reflect API to inspect and manipulate object properties. Next, it teaches how to create proxies, and use it to intercept and customize operations performed on objects.

Finally, it explains old modular programming techniques such as IIFE, CommonJS, AMD, and UMD, compares them with ES6 modules, and looks at how ES6 modules can increase the performance of websites when used.

Who this book is written for

If you are a JavaScript developer with basic development skills, and now want to learn about the latest ES6 features in order to develop better client-side programs with JavaScript, then this book is for you.

What you will learn from this book

- Explore the usage of the new syntaxes introduced by ES6
- Use the new prototype-based features introduced by ES6
- Execute ES6 in an old non-supported ES6 environment
- Write asynchronous code using Promise to facilitate the writing of better asynchronous code that is easier to read and maintain
- Create and understand the usage of iterators, iterables, and generators
- Get to know object-oriented programming and create objects using classes
- Build proxies using the ES6 proxy API and understand its uses
- Create JavaScript libraries using ES6 modules

Foreword by Jack Franklin, JavaScript Developer at GoCardless
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Knowing Your Library'
- A synopsis of the book’s content
- More information on Learning ECMAScript 6
About the Author

Narayan Prusty is a web and mobile app developer. He specializes in WordPress, HTML5, JavaScript, PHP, Solr, and Cordova. He has been learning and building applications using these technologies for many years.

He is the founder of QScutter.com, which provides courses on various application development topics and has more than 10,000 subscribers worldwide. His personal blog, http://www.QNimate.com, is among the top Intel XDK and WordPress blogs. He also works as a consultant and freelance developer for many companies worldwide.

Visit him online at http://www.twitter.com/narayanprusty.
ECMAScript is a scripting language standardized by Ecma International in the ECMA-262 specification and ISO/IEC 16262. Scripting languages such as JavaScript, JScript and ActionScript are supersets of ECMAScript. Although JavaScript, JScript, and ActionScript have more capabilities than ECMAScript, by defining more objects and methods, the core features of these languages are the same as ECMAScript.

ECMAScript 6 is the sixth version and the seventh edition of the ECMAScript language. In short, it is also called "ES6".

Although JavaScript is extremely powerful and flexible, it's often criticized for having unnecessary redundancy. Therefore, JavaScript developers often use abstractions such as CoffeeScript and Typescript, which provide easier syntax, powerful features, and compile to JavaScript. ES6 was introduced to improve JavaScript and ensure that developers no longer needed to use abstractions or other techniques to write quality code, which was becoming a lengthy process.

The ES6 features are inherited from other popular and abstracting languages, such as CoffeeScript. Therefore, ES6 language features behave the same way as they do in other languages, and are not new in the programming world, even if they are new in JavaScript.

This book provides explanations with examples for all the features of the new version of ECMAScript, ECMAScript 6. This book is about the JavaScript implementation of ECMAScript 6. All the features and examples in this book work on all the JavaScript environments, such as browsers, Node.js, Cordova, and so on.
What this book covers

Chapter 1, Playing with Syntax, introduces new ways of creating variables and function parameters. This chapter discusses the new objects and functions syntaxes in more depth.

Chapter 2, Knowing Your Library, introduces the new prototype-based methods for the existing objects.

Chapter 3, Using Iterators, shows the different types of iterators available in ES6, and how to create custom iterators. It also discusses the tail call optimization in ES6.

Chapter 4, Asynchronous Programming, illustrates how Promises can make it easier to write code that's executed asynchronously.

Chapter 5, Implementing the Reflect API, gives an in-depth guide to object reflection in ES6.

Chapter 6, Using Proxies, shows how to define the custom behavior the fundamental operations on the objects using the ES6 proxies.

Chapter 7, Walking You Through Classes, introduces the Object-oriented Programming using the ES6 classes. Concepts such as inheritance, constructors, abstraction, information hiding, and more are explained here.

Chapter 8, Modular Programming, explains different ways to create modules using JavaScript. Technologies such as the IIFE, CommonJS, AMD, UMD, and ES6 modules are covered.
ES6 has added lots of new properties and methods to built-in JavaScript objects, so that the programmer can do cumbersome tasks easily. These new functionalities aim to help the developers get rid of using hacks and error-prone techniques to do various operations related to numbers, strings, and arrays. In this chapter, we will look at all the new functionalities added by ES6 to the native objects.

In this chapter, we'll cover:

- The new properties and methods of the `Number` object
- Representing the numeric constants as binary or octal
- The new properties and methods of the `Math` object
- Creating the multiline strings and the new methods of the `String` object
- The new properties and methods of `Array` object
- What are Maps and Sets?
- Using array buffers and typed arrays
- The new properties and methods of `Object` object

**Working with numbers**

ES6 adds new ways of creating numbers and new properties to the `Number` object to make working with numbers easier. The `Number` object was enhanced in ES6 to make it easier to create mathematically rich applications, and prevent the common misconceptions that caused the errors. ES6 also provides new ways to do things that were already possible in ES5, such as representing the numeric constants as octal.
JavaScript represents the numbers as base 10 decimals. The numeric constants are, by default, interpreted as base 10 decimals.

The binary notation
In ES5, there was no native way to represent the numeric constants as binary. But in ES6, you can prefix the numeric constants using the `0b` token to make JavaScript interpret them as binary.

Here is an example:

```javascript
let a = 0b00001111;
let b = 15;

console.log(a === b);
console.log(a);
```

The output is as follows:

```
true
15
```

Here, `0b00001111` is a binary representation of 15, base 10 decimal.

The octal notation
In ES5, to represent a numeric constant as octal, we needed to prefix the numeric constant using `0`. For example, take a look at the following:

```javascript
var a = 017;
var b = 15;

console.log(a === b);
console.log(a);
```

The output is as the following:

```
true
15
```
But often, programmers new to JavaScript, get confused with the octal representations as the decimal number with 0 at the front. For example, they think 017 is same as 17. Therefore, to remove this confusion, ES6 lets us prefix the numeric constants using `0o` to make JavaScript interpret them as octal.

Here is an example to demonstrate this:

```javascript
let a = 0o17;
let b = 15;

console.log(a === b);
console.log(a);
```

The output is as follows:

```
true
15
```

### The Number.isUserInteger(number) method

JavaScript numbers are stored as the 64-bit, floating-point numbers. So the integers in JavaScript are the floating-point numbers without a decimal fraction, or a decimal fraction with all 0's.

In ES5, there was no built-in way to check whether a number is an integer or not. ES6 adds a new method to the Number object called `Number.isInteger()`, which takes a number and returns `true` or `false`, depending on whether the number is an integer or not.

Here is an example code:

```javascript
let a = 17.0;
let b = 1.2;

console.log(Number.isInteger(a));
console.log(Number.isInteger(b));
```

The output is as follows:

```
true
false
```
The Number.isNaN(value) method

In ES5, there was no way to check whether a variable holds the NaN value or not.

The global isNaN() function is used to check whether a value is a number or not. If the value is not a number, then it returns true, otherwise it returns false.

So ES6 introduced a new method for the Number object called as isNaN() to check whether a value is NaN or not. Here is an example, which demonstrates Number.isNaN() and also explains how it is different from the global isNaN() function:

```javascript
let a = "NaN";
let b = NaN;
let c = "hello";
let d = 12;

console.log(Number.isNaN(a));
console.log(Number.isNaN(b));
console.log(Number.isNaN(c));
console.log(Number.isNaN(d));

console.log(isNaN(a));
console.log(isNaN(b));
console.log(isNaN(c));
console.log(isNaN(d));
```

The output is as follows:

false
true
false
false
true
true
true
false

Here you can see that Number.isNaN() method returns true only if the passed value is exactly NaN.
You might ask, why not use == or the === operator instead of the `Number.isNaN(value)` method? The NaN value is the only value that is not equal to itself, that is, the expression `NaN==NaN` or `NaN===NaN` will return `false`.

**The Number.isFinite(number) method**

In ES5 there was no built-in way to check whether a value is a finite number.

The global `isFinite()` function takes a value and checks whether it's a finite number or not. But unfortunately, it also returns `true` for values that convert to a `Number` type.

So ES6 introduced the `Number.isFinite()` method, which resolves the issue of the `window.isFinite()` function. Here is an example to demonstrate this:

```javascript
console.log(isFinite(10));
console.log(isFinite(NaN));
console.log(isFinite(null));
console.log(isFinite([]));

console.log(Number.isFinite(10));
console.log(Number.isFinite(NaN));
console.log(Number.isFinite(null));
console.log(Number.isFinite([]));
```

The output is as follows:

```
true
false
true
true
true
false
false
false
false
```
The Number.isSafeInteger(number) method

The JavaScript numbers are stored as 64-bit floating-point numbers, following the international IEEE 754 standard. This format stores numbers in 64 bits, where the number (the fraction) is stored in 0 to 51 bits, the exponent in 52 to 62 bits, and the sign in the last bit.

So in JavaScript, safe integers are those numbers that are not needed to be rounded to some other integer to fit in the IEEE 754 representation. Mathematically, the numbers from \((-2^{53}-1)\) to \((2^{53}-1)\) are considered as safe integers.

Here is an example to demonstrate this:

```javascript
console.log(Number.isSafeInteger(156));
console.log(Number.isSafeInteger('1212'));
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MAX_SAFE_INTEGER + 1));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER));
console.log(Number.isSafeInteger(Number.MIN_SAFE_INTEGER - 1));
```

The output is as follows:

```javascript
true
false
true
false
true
false
```

Here, `Number.MAX_SAFE_INTEGER` and `Number.MIN_SAFE_INTEGER` are constant values, introduced in ES6, representing \((2^{53}-1)\) and \((-2^{53}-1)\) respectively.

The Number.EPSILON property

JavaScript uses such binary floating-point representation that the computers fail to accurately represent numbers like 0.1, 0.2, 0.3, and so on. When your code is executed, numbers like 0.1 are rounded to the nearest number in that format, which results in small rounding error.

Consider this example:

```javascript
console.log(0.1 + 0.2 == 0.3);
console.log(0.9 - 0.8 == 0.1);
console.log(0.1 + 0.2);
console.log(0.9 - 0.8);
```
The Number.EPSILON property was introduced in ES6, which has a value of approximately $2^{-52}$. This value represents a reasonable margin of error when comparing the floating-point numbers. Using this number, we can create a custom function to compare the floating-point numbers by ignoring the minimal rounding errors.

Here is an example code:

```javascript
function epsilonEqual(a, b)
{
    return Math.abs(a - b) < Number.EPSILON;
}

console.log(epsilonEqual(0.1 + 0.2, 0.3));
console.log(epsilonEqual(0.9 - 0.8, 0.1));
```

The output is as follows:

```
true
true
```

Here, `epsilonEqual()` is the custom function that we build to compare whether the two values are equal or not. Now, the output is as expected.

To learn more about this behavior of JavaScript and the floating-point arithmetic, visit [http://floating-point-gui.de/](http://floating-point-gui.de/).

**Doing Math**

ES6 adds a lot of new methods to the Math object, related to trigonometry, arithmetic, and miscellaneous. This lets the developers use native methods instead of external math libraries. Native methods are optimized for performance, and have better decimal precision.
Knowing Your Library

Trigonometry related operations
Here is an example code, which shows the entire trigonometry-related methods that are added to the Math object:

```javascript
console.log(Math.sinh(0));    // hyperbolic sine of a value
console.log(Math.cosh(0));    // hyperbolic cosine of a value
console.log(Math.tanh(0));    // hyperbolic tangent of a value
console.log(Math.asinh(0));   // inverse hyperbolic sine of a value
console.log(Math.acosh(1));   // inverse hyperbolic cosine of a value
console.log(Math.atanh(0));   // inverse hyperbolic tangent of a value
console.log(Math.hypot(2, 2, 1));  // Pythagoras theorem
```

The output is as follows:
```
0
1
0
0
0
0
3
```

Arithmetic related operations
Here is an example code, which shows the entire arithmetic related methods added to the Math object:

```javascript
console.log(Math.log2(16));   // log base 2
console.log(Math.log10(1000));  // log base 10
console.log(Math.log1p(0)); // same as log(1 + value)
console.log(Math.expm1(0)); // inverse of Math.log1p()
console.log(Math.cbrt(8));    // cube root of a value
```

The output is as follows:
```
4
3
0
0
2
```
Miscellaneous methods

ES6 adds some miscellaneous methods to the Math object. These methods are used for conversion and extracting information from the numbers.

The Math.imul(number1, number2) function

The Math.imul() takes two numbers as 32-bit integers and multiplies them. It returns the lower 32 bits of the result. This is the only native way to do 32-bit integer multiplication in JavaScript.

Here is an example to demonstrate this:

```
console.log(Math.imul(590, 5000000)); //32-bit integer multiplication
console.log(590 * 5000000); //64-bit floating-point multiplication
```

Output is:

```
-1344967296
2950000000
```

Here when multiplication was done it produced a number so large it couldn't be stored in 32 bits, therefore the lower bits were lost.

The Math.clz32(number) function

The Math.clz32() function returns the number of leading zero bits in the 32-bit representation of a number.

Here is an example to demonstrate this:

```
console.log(Math.clz32(7));
console.log(Math.clz32(1000));
console.log(Math.clz32(2950000000));
```

Output is:

```
29
22
3
```
Knowing Your Library

The Math.sign(number) function
The Math.sign() function returns the sign of a number, indicating whether the number is negative, positive or zero.

Here is an example to demonstrate this:

```javascript
console.log(Math.sign(11));
console.log(Math.sign(-11));
console.log(Math.sign(0));
```

Output is:

```
1
-1
0
```

From the preceding code, we can see that the Math.sign() function returns 1 if the number is positive, -1 if the number is negative, and 0 if the number is zero.

The Math.trunc(number) function
The Math.trunc() function returns the integer part of a number by removing any fractional digit. Here is an example to demonstrate this:

```javascript
console.log(Math.trunc(11.17));
console.log(Math.trunc(-1.112));
```

Output is:

```
11
-1
```

The Math.fround(number) function
The Math.fround() function rounds a number to a 32-bit floating point value. Here is an example to demonstrate this:

```javascript
console.log(Math.fround(0));
console.log(Math.fround(1));
console.log(Math.fround(1.137));
console.log(Math.fround(1.5));
```
Working with strings

ES6 provides new ways of creating strings and adds new properties to global String object and to its instances to make working with strings easier. Strings in JavaScript lacked features and capabilities when compared with programming languages such as Python and Ruby therefore ES6 enhanced strings to change that.

Before we get into new string features lets revise JavaScript's internal character encoding and escape sequences. In the Unicode character set every character is represented by a base 10 decimal number called a code point. A code unit is a fixed number of bits in memory to store a code point. An encoding schema determines the length of code unit. A code unit is 8 bits if the UTF-8 encoding schema is used or 16 bits if the UTF-16 encoding schema is used. If a code point doesn't fit in a code unit it is spilt into multiple code units, that is, multiple characters in sequence representing a single character.

JavaScript interpreters by default interpret JavaScript source code as sequence of UTF-16 code units. If source code is written in the UTF-8 encoding schema then there are various ways to tell the JavaScript interpreter to interpret it as sequence of UTF-8 code units. JavaScript strings are always a sequence of UTF-16 code points.

Any Unicode character with a code point less than 65536 can be escaped in a JavaScript string or source code using the hexadecimal value of its code point, prefixed with \\u. Escapes are six characters long. They require exactly four characters following \\u. If the hexadecimal character code is only one, two or three characters long, you'll need to pad it with leading zeroes. Here is an example to demonstrate this:

```javascript
var \u0061 = "\\u0061\\u0062\\u0063";
console.log(a); //Output is "abc"
```
Escaping larger code points

In ES5, for escaping a character that requires more than 16 bits for storing, we needed two Unicode escapes. For example, to add \u1F691 to a string we had to escape it this way:

```javascript
console.log("\uD83D\uDE91");
```

Here \uD83D and \uDE91 are called surrogate pairs. A surrogate pair is two Unicode characters when written in sequence represent another character.

In ES6 we can write it without surrogate pairs:

```javascript
console.log("\u{1F691}");
```

A string stores \u1F691 as \uD83D\uDE91, so length of the above string is still 2

The codePointAt(index) method

The codePointAt() method of a string returns a non-negative integer that is the code point of the character at the given index.

Here is an example to demonstrate this:

```javascript
console.log("\uD83D\uDE91".codePointAt(1));
console.log("\u{1F691}".codePointAt(1));
console.log("hello".codePointAt(2));
```

Output is:

```
56977
56977
1080
```

The String.fromCharCode(number1, ..., number 2) method

The fromCharCode() method of String object takes a sequence of code points and returns a string. Here is an example to demonstrate this:

```javascript
console.log(String.fromCharCode(0x61, 0x62, 0x63));
console.log("\u0061\u0062 " == String.fromCharCode(0x61, 0x62));
```
Output is:

abc
true

The `repeat(count)` method

The `repeat()` method of a string, constructs and returns a new string which contains the specified number of copies on which it was called, concatenated together. Here is an example to demonstrate this:

```javascript
console.log("a".repeat(6));  // Output "aaaaaa"
```

The `includes(string, index)` method

The `includes()` method is used to find whether one string may be found in another string, returning `true` or `false` as appropriate. Here is an example to demonstrate this:

```javascript
var str = "Hi, I am a JS Developer";
console.log(str.includes("JS"));  // Output "true"
```

It takes an optional second parameter representing the position in the string at which to begin searching. Here is an example to demonstrate this:

```javascript
var str = "Hi, I am a JS Developer";
console.log(str.includes("JS", 13));  // Output "false"
```

The `startsWith(string, index)` method

The `startsWith()` method is used to find whether a string begins with the characters of another string, returning `true` or `false` as appropriate. Here is an example to demonstrate this:

```javascript
var str = "Hi, I am a JS Developer";
console.log(str.startsWith("Hi, I am"));  // Output "true"
```

It takes an optional second parameter representing the position in the string at which to begin searching. Here is an example to demonstrate this:

```javascript
var str = "Hi, I am a JS Developer";
console.log(str.startsWith("JS Developer", 11));  // Output "true"
```
The `endsWith(string, index)` function

The `endsWith()` method is used to find whether a string ends with the characters of another string, returning true or false as appropriate. It also takes an optional second parameter representing the position in the string that is assumed as the end of the string. Here is an example to demonstrate this:

```javascript
var str = "Hi, I am a JS Developer";
console.log(str.endsWith("JS Developer"));  //Output "true"
console.log(str.endsWith("JS", 13));        //Output "true"
```

Normalization

Normalization is simply the process of searching and standardizing code points without changing the meaning of the string.

There are also different forms of normalization: NFC, NFD, NFKC and NFKD.

Let's understand Unicode string normalization by an example use case:

A case study

There are many Unicode characters that can be stored in 16 bits and can also be represented using a surrogate pair. For example, 'é' character can be escaped two ways:

```javascript
console.log("\u00E9");  //output 'é'
console.log("e\u0301"); //output 'é'
```

The problem is when applying the `==` operator, iterating or finding length you will get an unexpected result. Here is an example to demonstrate this:

```javascript
var a = "\u00E9";
var b = "e\u0301";

console.log(a == b);
console.log(a.length);
console.log(b.length);

for(let i = 0; i<a.length; i++)
{
    console.log(a[i]);
}
for(let i = 0; i<b.length; i++)
{
    console.log(b[i]);
}

Output is:
false
1
2
é
éœ

Here both the strings display the same way but when we do various string operations on them we get different results.

The length property ignores surrogate pairs and assumes every 16-bit to be single character. The == operator matches the binary bits therefore it also ignores surrogate pairs. The [ ] operator also assumes every 16-bit to be an index therefore ignoring surrogate pairs.

In this case to solve the problems we need to convert the surrogate pairs to 16-bit character representation. This process is called as normalization. To do this ES6 provides a normalize() function. Here is an example to demonstrate this:

var a = "\u00E9".normalize();
var b = "e\u0301".normalize();

console.log(a == b);
console.log(a.length);
console.log(b.length);

for(let i = 0; i<a.length; i++)
{
    console.log(a[i]);
}

for(let i = 0; i<b.length; i++)
{
    console.log(b[i]);
}
Output is:

true
1
1
é
é

Here the output is as expected. `normalize()` returns the normalized version of the string. `normalize()` uses NFC form by default.

Normalization is not just done in the case of surrogate pairs; there are many other cases.

```
The Normalized version of a string is not made for displaying to the user; it's used for comparing and searching in strings.
```

To learn more about Unicode string normalization and normalization forms visit http://www.unicode.org/reports/tr15/

**Template strings**

Template strings is just a new literal for creating strings that makes various things easier. They provide features such as embedded expressions, multi-line strings, string interpolation, string formatting, string tagging, and so on. They are always processed and converted to a normal JavaScript string on runtime therefore they can be used wherever we use normal strings.

Template strings are written using back ticks instead of single or double quotes. Here is an example of a simple template string:

```javascript
let str1 = `hello!!!`; //template string
let str2 = "hello!!!";

console.log(str1 === str2); //output "true"
```

**Expressions**

In ES5, to embed expressions within normal strings you would do something like this:

```javascript
Var a = 20;
Var b = 10;
```
Var c = "JavaScript";
Var str = "My age is " + (a + b) + " and I love " + c;

console.log(str);

Output is:

My age is 30 and I love JavaScript

In ES6, template strings make it much easier to embed expressions in strings. Template strings can contain expressions in them. The expressions are placed in placeholders indicated by dollar sign and curly brackets, that is, ${expressions}. The resolved value of expressions in the placeholders and the text between them are passed to a function for resolving the template string to a normal string. The default function just concatenates the parts into a single string. If we use a custom function to process the string parts then the template string is called as a tagged template string and the custom function is called as tag function.

Here is an example which shows how to embed expressions in a template strings:

```javascript
let a = 20;
let b = 10;
let c = "JavaScript";
let str = `My age is ${a+b} and I love ${c}`;

console.log(str);
```

Output is:

My age is 30 and I love JavaScript

Let's create a tagged template string, that is, process the string using a tag function. Let's implement the tag function to do the same thing as the default function. Here is an example to demonstrate this:

```javascript
let tag = function(strings, ...values)
{
    let result = "";

    for(let i = 0; i<strings.length; i++)
    {
        result += strings[i];
        if(i<values.length)
        {
            result += values[i];
        }
    }
    return result;
}

let a = 20;
let b = 10;
let c = "JavaScript";
let str = `My age is ${a+b} and I love ${c}`;

console.log(tag(strings, a, b, c));
```

Output is:

My age is 30 and I love JavaScript

Let's create a tagged template string, that is, process the string using a tag function. Let's implement the tag function to do the same thing as the default function. Here is an example to demonstrate this:

```javascript
let tag = function(strings, ...values)
{
    let result = "";

    for(let i = 0; i<strings.length; i++)
    {
        result += strings[i];
        if(i<values.length)
        {
            result += values[i];
        }
    }
    return result;
}

let a = 20;
let b = 10;
let c = "JavaScript";
let str = `My age is ${a+b} and I love ${c}`;

console.log(tag(strings, a, b, c));
```
let a = 20;
let b = 10;
let c = "JavaScript";
let str = tag `My age is ${a+b} and I love ${c}`;

console.log(str);

Output is:

My age is 30 and I love JavaScript

Here our tag function's name is tag but you can name it anything else. The custom function takes two parameters, that is, the first parameter is an array of string literals of the template string and the second parameter is an array of resolved values of the expressions. The second parameter is passed as multiple arguments therefore we use the rest argument.

**Multiline strings**

Template strings provide a new way to create strings that contain multiple lines of text.

In ES5, we need to use \n new line character to add new line breaks. Here is an example to demonstrate this:

```javascript
console.log("1\n2\n3");
```

Output is

```
1
2
3
```

In ES6, using **multiline** string we can simply write:

```javascript
console.log(`1
2
3`);
```
Chapter 2

In the above code we simply included new lines where we needed to place \n. While converting the template string to normal string the new lines are converted to \n.

**Raw strings**

A raw string is a normal string in which escaped characters aren't interpreted.

We can create a raw string using a template string. We can get raw version of a template string use String.raw tag function. Here is an example to demonstrate this:

```javascript
let s = String.raw `xy\n{ 1 + 1 }z`;
console.log(s);
```

Output is:

```
xy\n2z
```

Here \n is not interpreted as new line character instead of its two characters, that is, \ and n. Length of variable s would be 6.

If you create a tagged function and you want to return the raw string then use raw property of the first argument. raw property is an array, which holds raw versions of the strings of the first argument. Here is an example to demonstrate this:

```javascript
let tag = function(strings, ...values)
{
    return strings.raw[0]
};

letstr = tag `Hello \n World!!!`;

console.log(str);
```

Output is:

```
Hello \n World!!!
```
Arrays
ES6 adds new properties to the global Array object and to its instances to make working with arrays easier. Arrays in JavaScript lacked features and capabilities when compared with programming languages such as Python and Ruby therefore ES6 enhanced arrays to change that.

The Array.from(iterable, mapFunc, this) method
The `Array.from()` method creates a new array instance from an iterable object. The first argument is a reference to the iterable object. The second argument is optional and is a callback (known as Map function) that is called for every element of the iterable object. The third argument is also optional and is the value of `this` inside the Map function.

Here is an example to demonstrate this:

```javascript
let str = "0123";
let obj = {number: 1};
let arr = Array.from(str, function(value){
    return parseInt(value) + this.number;
}, obj);

console.log(arr);
```

Output is:

```
1, 2, 3, 4
```

The Array.of(values...) method
The `Array.of()` method is an alternative to the `Array` constructor for creating arrays. When using `Array` constructor if we pass only one argument, that too a number, then `Array` constructor constructs an empty array with array length property equal to the passed number instead of creating an array of one element with that number in it. Therefore the `Array.of()` method was introduced to resolve this issue.
Here is an example to demonstrate this:

```javascript
let arr1 = new Array(2);
let arr2 = new Array.of(2);

console.log(arr1[0], arr1.length);
console.log(arr2[0], arr2.length);
```

Output is:

```
undefined 2
2 1
```

You should use `Array.of()` instead of `Array` constructor when you are constructing a new array instance dynamically, that is, when you don’t know the type of values and the number of elements.

### The fill(value, startIndex, endIndex) method

The `fill()` method of an array fills all the elements of the array from the `startIndex` to an `endIndex` (not including `endIndex`) with a given value. Remember that `startIndex` and `endIndex` arguments are optional; therefore if they are not provided then the whole array is filled with the given value. If only `startIndex` is provided then `endIndex` defaults to length of array minus 1.

If `startIndex` is negative then it’s treated as length of array plus `startIndex`. If `endIndex` is negative, it is treated as length of array plus `endIndex`.

Here is an example to demonstrate this:

```javascript
let arr1 = [1, 2, 3, 4];
let arr2 = [1, 2, 3, 4];
let arr3 = [1, 2, 3, 4];
let arr4 = [1, 2, 3, 4];
let arr5 = [1, 2, 3, 4];

arr1.fill(5);
arr2.fill(5, 1, 2);
arr3.fill(5, 1, 3);
arr4.fill(5, -3, 2);
arr5.fill(5, 0, -2);
```
The `find(testingFunc, this)` method

The `find()` method of an array returns an array element, if it satisfies the provided testing function. Otherwise it returns `undefined`.

The `find()` method takes two arguments, that is, the first argument is the testing function and the second argument is the value of `this` in the testing function. The second argument is optional.

The testing function has three parameters: the first parameter is the array element being processed, second parameter is the index of the current element being processed and third parameter is the array on which `find()` is called upon.

The testing function needs to return `true` to satisfy a value. The `find()` method returns the first element which satisfies the provided testing function.

Here is an example to demonstrate the `find()` method:

```javascript
var x = 12;
var arr = [11, 12, 13];
var result = arr.find(function(value, index, array){
    if(value == this){
        return true;
    }
}, x);

console.log(result); //Output "12"
```
The findIndex(testingFunc, this) method

The `findIndex()` method is similar to the `find()` method. The `findIndex()` method returns the index of the satisfied array element instead of the element itself.

```javascript
let x = 12;
let arr = [11, 12, 13];
let result = arr.findIndex(function(value, index, array){
  if(value == this)
  {
    return true;
  }
}, x);

console.log(result); // Output "1"
```

The copyWithin(targetIndex, startIndex, endIndex) function

The `copyWithin()` method of an array is used to copy the sequence of values of the array to a different position in the array.

The `copyWithin()` method takes three arguments: first argument represents the target index where to copy elements to, second argument represents the index position where to start copying from and the third argument represents the index, that is, where to actually end copying elements.

The third argument is optional and if not provided then it defaults to `length-1` where `length` is the length of the array. If `startIndex` is negative then it's calculated as `length+startIndex`. Similarly if `endIndex` is negative then it's calculated as `length+endIndex`.

Here is an example to demonstrate this:

```javascript
let arr1 = [1, 2, 3, 4, 5];
let arr2 = [1, 2, 3, 4, 5];
let arr3 = [1, 2, 3, 4, 5];
let arr4 = [1, 2, 3, 4, 5];

arr1.copyWithin(1, 2, 4);
arr2.copyWithin(0, 1);
```
Knowing Your Library

```javascript
arr3.copyWithin(1, -2);
arr4.copyWithin(1, -2, -1);

console.log(arr1);
console.log(arr2);
console.log(arr3);
console.log(arr4);
```

Output is:

1,3,4,4,5
2,3,4,5,5
1,4,5,4,5
1,4,3,4,5

The entries(), keys(), and values() method

The entries() method of an array returns an iterable object that contains key/value pairs for each index of the array. Similarly the keys() method of an array returns an iterable object that contains keys for each of the indexes in the array. Similarly, the values() method of an array returns an iterable object that contains values of the array.

The iterable object returned by the entries() method stores the key/value pairs in the form of arrays.

The iterable object returned by these functions is not an array.

Here is an example to demonstrate this:

```javascript
let arr = ['a', 'b', 'c'];
let entries = arr.entries();
let keys = arr.keys();
let values = arr.values();

console.log(...entries);
console.log(...keys);
console.log(...values);
```

Output is:

0,a 1,b 2,c
0 1 2
a b c
Collections

A collection is any object that stores multiple elements as a single unit. ES6 introduced various new collection objects to provide better ways of storing and organizing data.

The array is the only collection object available in ES5. ES6 introduces array buffers, typed arrays, Sets, and Maps, which are built in collection objects.

Let's see the different collection objects provided by ES6.

Array buffers

Elements of arrays can be of any type such as strings, numbers, objects, and so on. Arrays can grow dynamically. The problem with arrays is that they are slow in terms of execution time, and occupy more memory. This causes issues while developing applications that require too much computation and deal with large amount of numbers. Therefore array buffers were introduced to tackle this issue.

An array buffer is a collection of 8-bit blocks in memory. Every block is an array buffer element. The size of an array buffer needs to be decided while creating, it therefore it cannot grow dynamically. Array buffers can only store numbers. All blocks are initialized to number 0 on creation of an array buffer.

An array buffer object is created using ArrayBuffer constructor.

```javascript
let buffer = new ArrayBuffer(80); //80 bytes size
```

Reading from and writing values into an array buffer object can be done using a DataView object. It's not compulsory that only 8 bits are used to represent a number. We can use 8, 16, 32 and 64 bits to represent a number. Here is an example, which shows how to create a DataView object and read/write to an ArrayBuffer object:

```javascript
let buffer = new ArrayBuffer(80);
let view = new DataView(buffer);

view.setInt32(8, 22, false);

var number = view.getInt32(8, false);

console.log(number); //Output "22"
```
Knowing Your Library

Here we created a DataView object using DataView constructor. A DataView object provides several methods to read and write numbers into an array buffer object. Here we used.setInt32() method which uses 32 bits to store a provided number.

All the methods of a DataView object that are used to write data to an array buffer object take three arguments. First argument represents the offset, that is, the byte we want to write the number to. Second argument is the number to be stored. And third argument is a Boolean type, which represents the endian of the number, such as false represents a big-endian.

Similarly all the methods of a DataView object that are used to read data of an array buffer object take two arguments. First argument is the offset and second argument represents the endian used.

Here are other functions for storing numbers provided by a DataView object:

- **setInt8**: Uses 8 bits to store a number. It takes a signed integer (-ve or +ve).
- **setUint8**: Uses 8 bits to store a number. It takes an unsigned integer (+ve).
- **setInt16**: Uses 16 bits to store a number. It takes a signed integer.
- **setUint16**: Uses 16 bits to store a number. It takes an unsigned integer.
- **setInt32**: Uses 32 bits to store a number. It takes a signed integer.
- **setUint32**: Uses 32 bits to store a number. It takes an unsigned integer.
- **setFloat32**: Uses 32 bits to store a number. It takes a signed decimal number.
- **setFloat64**: Uses 64 bits to store a number. It takes a signed decimal number.

Here are other functions for retrieving stored numbers by a DataView object:

- **getInt8**: Reads 8 bits. Returns signed integer number.
- **getUint8**: Reads 8 bits. Returns unsigned integer number.
- **getInt16**: Reads 16 bits. Returns signed integer number.
- **getUint16**: Reads 16 bits. Returns unsigned integer number.
- **getInt32**: Reads 32 bits. Returns signed integer number.
- **getUint32**: Reads 32 bits. Returns unsigned integer number.
- **getFloat32**: Reads 32 bits. Returns signed decimal number.
- **getFloat64**: Reads 64 bits. Returns signed decimal number.
Typed arrays

We saw how to read and write numbers in array buffers. But the method was very cumbersome because we had to call a function every time. Typed arrays let us read and write to an array buffer object just like we do for normal arrays.

A typed array acts like a wrapper for an array buffer object and treats data of an array buffer object as a sequence of $n$-bit numbers. The $n$ value depends on how we created the typed array.

Here is the code example that demonstrates how to create an array buffer object and read/write to it using a typed array:

```javascript
var buffer = new ArrayBuffer(80);
var typed_array = new Float64Array(buffer);
typed_array[4] = 11;

console.log(typed_array.length);
console.log(typed_array[4]);
```

Output is:

```
10
11
```

Here we created typed array using the `Float64Array` constructor, it therefore treats data in the array buffer as a sequence of 64-bit signed decimal numbers. Here the array buffer object size was 640 bits therefore only 10 64-bit numbers can be stored.

Similarly, there are other typed array constructors to represent data in an array buffer as a sequence of different bit numbers. Here is the list:

- **Int8Array**: Treats as 8-bit signed integers
- **Uint8Array**: Treats as 8-bit unsigned integers
- **Int16Array**: Treats as 16-bit signed integers
- **Uint16Array**: Treats as 16-bit unsigned integers
- **Int32Array**: Treats as 32-bit signed integers
- **Uint32Array**: Treats as 32-bit unsigned integers
- **Float32Array**: Treats as 32-bit signed decimal number
- **Float64Array**: Treats as 64-bit signed decimal number
Typed arrays provide all the methods that are also provided by normal JavaScript arrays. They also implement the iterable protocol therefore they can be used as an iterable object.

**Set**

A **Set** is a collection of *unique* values of any data type. The values in a Set are arranged in insertion order. A Set is created using `Set` constructor. Here is an example:

```javascript
let set1 = new Set();
let set2 = new Set("Hello!!!");
```

Here `set1` is an empty Set. Whereas `set2` was created using values of an iterable object, that is, characters of a string and the string was not empty therefore `set2` is non-empty.

Here is example code, which demonstrates various operations that can be done on a Set:

```javascript
let set = new Set("Hello!!!");
set.add(12); //add 12
console.log(set.has("!")); //check if value exists
console.log(set.size);
set.delete(12); //delete 12
console.log(...set);
set.clear(); //delete all values
```

Output is:

```
true
6
H e l o !
```

Here we added nine items to the `set` object but the `size` was only six because `Set` automatically deletes duplicate values. The characters `l` and `i` were repeated multiple times.
The Set object also implements the iterable protocol so they can be used as an iterable object.

Sets are used when you want to maintain a collection of values and check if a value exists instead of retrieving a value. For example: Sets can be used as an alternative to an array if you only use the `indexOf()` method of the array in your code to check if an value exists.

**WeakSet**

Here are the differences between Set and WeakSet objects:

- A Set can store primitive types and object references whereas a WeakSet object can only store object references
- One of the important features of the WeakSet object is that if there is no other reference to an object stored in a WeakSet object then they are garbage collected
- Lastly a WeakSet object is not enumerable, that is, you cannot find its size; it also doesn't implement iterable protocol

Apart from these three differences it behaves exactly the same way as a Set. Everything else apart from these three differences is same between a Set and WeakSet object.

A WeakSet object is created using `WeakSet` constructor. You cannot pass an iterable object as an argument to WeakSet object.

Here is an example to demonstrate WeakSet:

```javascript
let weakset = new WeakSet();

(function(){
  let a = {};
  weakset.add(a);
})() //here 'a' is garbage collected from weakset

console.log(weakset.size); //output "undefined"
console.log(...weakset); //Exception is thrown
weakset.clear(); //Exception, no such function
```
Map

A Map is a collection of key/value pairs. Keys and values of a Map can be of any data type. The key/value pairs are arranged in the insertion order. A Map object is created using the `Map` constructor.

Here is an example, which demonstrates how to create a Map object and do various operations on it:

```javascript
let map = new Map();
let o = {n: 1};

map.set(o, "A"); //add
map.set("2", 9);

console.log(map.has("2")); //check if key exists
console.log(map.get(o)); //retrieve value associated with key
console.log(...map);

map.delete("2"); //delete key and associated value
map.clear(); //delete everything

//create a map from iterable object
let map_1 = new Map([ [1, 2], [4, 5] ]);;

console.log(map_1.size); //number of keys
```

Output is:

```
true
A
[object Object],A 2,9
2
```

While creating a Map object from an iterable object we need to make sure that the values returned by the iterable object are arrays, each of length 2 i.e., index 0 is the key and index 1 is the value.

If we try to add a key that already exists then it's overwritten. The Map objects also implement the iterable protocol and can therefore also be used as an iterable object. While iterating Maps using the iterable protocol, they return arrays with key/value pairs as you can see in the preceding example.
WeakMap

Here are the differences between the Map and the WeakMap objects:

- Keys of a Map object can be of primitive types or object references but keys in WeakMap object can only be object references
- One of the important features of the WeakMap object is that if there is no other reference to an object that is referenced by a key then the key is garbage collected.
- Lastly the WeakMap object is not enumerable, that is, you cannot find its size and it doesn't implement iterable protocol.

Everything else, apart from these three differences is similar between the Map and the WeakMap object.

A WeakMap is created using WeakMap constructor. Here is an example that demonstrates its usage:

```javascript
let weakmap = new WeakMap();

(function(){
  let o = {n: 1};
  weakmap.set(o, "A");
})();

//here 'o' key is garbage collected
let s = {m: 1};

weakmap.set(s, "B");

console.log(weakmap.get(s));
console.log(...weakmap); //exception thrown

weakmap.delete(s);
weakmap.clear(); //Exception, no such function

let weakmap_1 = new WeakMap([[{}, 2], [{}, 5]]); //this works

console.log(weakmap_1.size); //undefined
```
Object

ES6 standardizes the __proto__ property of an object and adds new properties to the global Object object.

The __proto__ property

JavaScript objects have an internal [[prototype]] property that references the object's prototype, that is, the object it inherits. To read the property we had to use Object.getPrototypeOf() and to create a new object with a given prototype we had to use the Object.create() method. A [[prototype]] property cannot be directly read or be modified.

Inheriting was cumbersome due to the nature of the [[prototype]] property, therefore some browsers added a special __proto__ property in objects, which is an accessor property that exposes the internal [[prototype]] property and makes working with prototypes easier. The __proto__ property was not standardized in ES5 but due to its popularity it was standardized in ES6.

Here is an example to demonstrate this:

```javascript
// In ES5
var x = {x: 12};
var y = Object.create(x, {y: {value: 13}});

console.log(y.x); // Output "12"
console.log(y.y); // Output "13"

// In ES6
let a = {a: 12, __proto__: {b: 13}};
console.log(a.a); // Output "12"
console.log(a.b); // Output "13"
```

The Object.is(value1, value2) method

The Object.is() method determines whether two values are equal or not. It is similar to the === operator but there are some special cases for the Object.is() method. Here is an example that demonstrates the special cases:

```javascript
console.log(Object.is(0, -0));
console.log(0 === -0);
console.log(Object.is(NaN, 0/0));
```
console.log(NaN === 0/0);
console.log(Object.is(NaN, NaN));
console.log(NaN === NaN);

Output is:
false
true
true
false
true
false

**The Object.setPrototypeOf(object, prototype) method**
The `Object.setPrototypeOf()` method is just another way to assign the `[[prototype]]` property of an object. Here is an example to demonstrate this:

```javascript
let x = {x: 12};
let y = {y: 13};
Object.setPrototypeOf(y, x)

console.log(y.x); //Output "12"
console.log(y.y); //Output "13"
```

**The Object.assign(targetObj, sourceObjs...) method**
The `Object.assign()` method is used to copy the values of all enumerable own properties from one or more source objects to a target object. This method will return the target object.

Here is an example which demonstrates this:

```javascript
let x = {x: 12};
let y = {y: 13, __proto__: x};
let z = {z: 14, get b() {return 2;}, q: {}};
Object.defineProperty(z, "z", {enumerable: false});
let m = {};
```
Knowing Your Library

```javascript
Object.assign(m, y, z);

console.log(m.y);
console.log(m.z);
console.log(m.b);
console.log(m.x);
console.log(m.q == z.q);
```

Output is:

```
13
undefined
2
undefined
true
```

Here is a list of important things to keep in mind while using the `Object.assign()` method:

- It invokes getters on the sources and setters on the target.
- It just assigns values of the properties of source to the new or existing properties of target.
- It doesn't copy the `[[prototype]]` property of sources.
- JavaScript property names can be strings or symbols. `Object.assign()` copies both.
- Property definitions are not copied from sources therefore you need to use `Object.getOwnPropertyDescriptor()` instead.
- It ignores copying keys with `null` and `undefined` values.

**Summary**

In this chapter we learned new features added by ES6 for working with numbers, strings, arrays and objects. We saw how arrays impact performance in math-rich applications and how the array buffers can be used instead. We also walked through the new collection objects provided by ES6.

In next chapter, we will learn about Symbols and Iteration protocol, and we will discover `yield` keyword and generators also.
Where to buy this book

You can buy Learning ECMAScript 6 from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.