ES6 ECMAScript 6

ECMAScript 6

ECMAScript (ES) is a scripting language specification created to standardize JavaScript.

The Sixth Edition, initially known as **ECMAScript 6** (ES6) and later renamed to **ECMAScript 2015**, adds significant new syntax for writing complex applications, including classes and modules, iterators and for/of loops, generators, arrow functions, binary data, typed arrays, collections (maps, sets and weak maps), promises, number and math enhancements, reflection, and proxies.

In other words, ES6 is a superset of JavaScript (ES5). The reason that ES6 became so popular is that it introduced new conventions and OOP concepts such as classes.

> In this module, we cover the most important additions to **ES6**.
> So, let's jump right in!

var & let

In ES6 we have three ways of declaring variables:

```
var a = 10;
const b = 'hello';
let c = true;
```

The type of declaration used depends on the necessary **scope**. **Scope** is the fundamental concept in all programming languages that defines the visibility of a variable.

var & **let**
Unlike the **var** keyword, which defines a variable globally, or locally to an entire function regardless of block scope, **let** allows you to declare variables that are limited in scope to the block, statement, or expression in which they are used.
For example:

```
if (true) {
  let name = 'Jack';
}
alert(name); //generates an error
```

Try It Yourself

In this case, the **name** variable is accessible only in the scope of the **if** statement because it was declared as **let**.

To demonstrate the difference in scope between **var** and **let**, consider this example:

```
function varTest() {
  var x = 1;
  if (true) {
    var x = 2;  // same variable
    console.log(x);  // 2
  }
  console.log(x);  // 2
}
```

```
function letTest() {
  let x = 1;
  if (true) {
    let x = 2;  // different variable
    console.log(x);  // 2
  }
  console.log(x);  // 1
}
```

Try It Yourself

One of the best uses for **let** is in loops:

```
for (let i = 0; i < 3; i++) {
  document.write(i);
}
```

Try It Yourself

Here, the **i** variable is accessible only within the scope of the **for** loop, where it is needed.

> **let** is not subject to **Variable Hoisting**, which means that **let** declarations do not move to the top of the current execution context.

---

const

const variables have the same scope as variables declared using **let**. The difference is that const variables are **immutable** - they are not allowed to be reassigned.
For example, the following generates an exception:

```
const a = 'Hello';
a = 'Bye';
```

Try It Yourself

> **const** is not subject to **Variable Hoisting** too, which means that **const** declarations do not move to the top of the current execution context.
> Also note that ES6 code will run only in browsers that support it. Older devices and browsers that do not support ES6 will return a syntax error.

Template Literals in ES6

Template literals are a way to output variables in the string.
Prior to ES6 we had to break the string, for example:

```
let name = 'David';
let msg = 'Welcome ' + name + '!';
console.log(msg);
```

Try It Yourself

ES6 introduces a new way of outputting variable values in strings. The same code above can be rewritten as:

```
let name = 'David';
let msg = `Welcome ${name}!`;
console.log(msg);
```

Try It Yourself

Notice, that template literals are enclosed by the **backtick** (` `) character instead of double or single quotes.
The **${expression}** is a placeholder, and can include any expression, which will get evaluated and inserted into the template literal.

For example:

```
let a = 8;
let b = 34;
let msg = `The sum is ${a+b}`;
console.log(msg);
```

Try It Yourself

To escape a backtick in a template literal, put a backslash \ before the backtick.

Loops in ECMAScript 6

In JavaScript we commonly use the **for** loop to iterate over values in a list:

```
let arr = [1, 2, 3];
for (let k = 0; k < arr.length; k++) {
  console.log(arr[k]);
}
```

Try It Yourself

The **for...in** loop is intended for iterating over the enumerable keys of an object.
For example:

```
let obj = {a: 1, b: 2, c: 3};
for (let v in obj) {
  console.log(v);
}
```

Try It Yourself

The **for...in** loop should **NOT** be used to iterate over arrays because, depending on the JavaScript engine, it could iterate in an arbitrary order. Also, the iterating variable is a **string**, not a number, so if you try to do any math with the variable, you'll be performing string concatenation instead of addition.

ES6 introduces the new **for...of** loop, which creates a loop iterating over iterable objects.
For example:

```
let list = ["x", "y", "z"];
for (let val of list) {
  console.log(val);
}
```

Try It Yourself

During each iteration the **val** variable is assigned the corresponding element in the list.

The **for...of** loop works for other iterable objects as well, including **strings**:

```
for (let ch of "Hello") {
  console.log(ch);
}
```

Try It Yourself

The **for...of** loop also works on the newly introduced collections (**Map**, **Set**, **WeakMap**, and **WeakSet**). We will learn about them in the upcoming lessons.
Note that ES6 code will run only in browsers that support it. Older devices and browsers that do not support ES6 will return a syntax error.

Functions in ECMAScript 6

Prior to ES6, a JavaScript function was defined like this:

```
function add(x, y) {
  var sum = x+y;
  console.log(sum);
}
```

Try It Yourself

ES6 introduces a new syntax for writing functions. The same function from above can be written as:

```
const add = (x, y) => {
  let sum = x + y;
  console.log(sum);
}
```

Try It Yourself

This new syntax is quite handy when you just need a simple function with one argument.
You can skip typing **function** and **return**, as well as some parentheses and braces.
For example:

```
const greet = x => "Welcome " + x;
```

Try It Yourself

The code above defines a function named **greet** that has one argument and returns a message.

If there are no parameters, an empty pair of parentheses should be used, as in

```
const x = () => alert("Hi");
```

Try It Yourself

The syntax is very useful for inline functions. For example, let's say we have an array, and for each element of the array we need to execute a function. We use the **forEach** method of the array to call a function for each element:

```
var arr = [2, 3, 7, 8];

arr.forEach(function(el) {
  console.log(el * 2);
});
```

Try It Yourself

However, in ES6, the code above can be rewritten as following:

```
const arr = [2, 3, 7, 8];

arr.forEach(v => {
  console.log(v * 2);
});
```

Try It Yourself

> The code is shorter and looks pretty nice, doesn't it? :)

Default Parameters in ES6

In ES6, we can put the default values right in the signature of the functions.
For example:

```
function test(a, b = 3, c = 42) {
  return a + b + c;
}
console.log(test(5)); //50
```

Try It Yourself

And here's an example of an arrow function with default parameters:

```
const test = (a, b = 3, c = 42) => {
  return a + b + c;
}
console.log(test(5)); //50
```

> Default value expressions are evaluated at function call time from left to right. This also means that default expressions can use the values of previously-filled parameters.

ES6 Objects

JavaScript variables can be **Object** data types that contain many values called **properties**.
An object can also have properties that are function definitions called **methods** for performing actions on the object.
ES6 introduces **shorthand** notations and **computed** property names that make declaring and using objects easier to understand.

The new method definition shorthand does not require the colon (:) or **function** keyword, as in the **grow** function of the **tree** object declaration:

```
let tree = {
  height: 10,
  color: 'green',
  grow() {
    this.height += 2;
  }
};
tree.grow();
console.log(tree.height); // 12
```

Try It Yourself

You can also use a property value shorthand when initializing properties with a variable by the same name.
For example, properties **height** and **health** are being initialized with variables named **height** and **health**:

```
let height = 5;
let health = 100;

let athlete = {
  height,
  health
};
```

Try It Yourself

When creating an object by using duplicate property names, the last property will overwrite the prior ones of the same name.
For Example:

```
var a = {x: 1, x: 2, x: 3, x: 4};
```

Try It Yourself

Duplicate property names generated a **SyntaxError** in ES5 when using strict mode. However, ES6 removed this limitation.

Computed Property Names

With ES6, you can now use **computed property** names. Using the square bracket notation [], we can use an expression for a property name, including concatenating strings. This can be useful in cases where we want to create certain objects based on user data (e.g. id, email, and so on).

Here are three examples:
Example 1:

```
let prop = 'name';
let id = '1234';
let mobile = '08923';

let user = {
  [prop]: 'Jack',
  [`user_${id}`]: `${mobile}`
};
```

Try It Yourself

Example 2:

```
var i = 0;
var a = {
  ['foo' + ++i]: i,
  ['foo' + ++i]: i,
  ['foo' + ++i]: i
};
```

Try It Yourself

Example 3:

```
var param = 'size';
var config = {
  [param]: 12,
  ['mobile' + param.charAt(0).toUpperCase() + param.slice(1)]: 4
};
console.log(config);
```

Try It Yourself

It is very useful when you need to create custom objects based on some variables.

Object.assign() in ES6

ES6 adds a new **Object** method **assign()** that allows us to combine multiple sources into one target to create a single new object.
Object.assign() is also useful for creating a duplicate of an existing object.

Let's look at the following example to see how to combine objects:

```
let person = {
  name: 'Jack',
  age: 18,
  sex: 'male'
};
let student = {
  name: 'Bob',
  age: 20,
  xp: '2'
};
let newStudent = Object.assign({}, person, student);
```

Try It Yourself

Here we used **Object.assign()** where the first parameter is the **target** object you want to apply new properties to.
Every parameter after the first will be used as **sources** for the target. There are no limitations on the number of source parameters. However, order is important because properties in the second parameter will be overridden by properties of the same name in third parameter, and so on.

In the example above, we used a new object **{}** as the target and used two objects as sources.

> Try changing the order of second and third parameters to see what happens to the result.

Now, let's see how we can use **assign()** to create a duplicate object without creating a reference (mutating) to the base object.
In the following example, assignment was used to try to generate a new object. However, using = creates a reference to the base object. Because of this reference, changes intended for a new object mutate the original object:

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = person; //  newPerson references person
newPerson.name = 'Bob';

console.log(person.name); // Bob
console.log(newPerson.name); // Bob
```

Try It Yourself

To avoid this (mutations), use **Object.assign()** to create a new object.
For example:

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = Object.assign({}, person);
newPerson.name = 'Bob';

console.log(person.name); // Jack
console.log(newPerson.name); // Bob
```

Try It Yourself

Finally, you can assign a value to an object property in the **Object.assign()** statement.
For example:

```
let person = {
  name: 'Jack',
  age: 18
};

let newPerson = Object.assign({}, person, {name: 'Bob'});
```

Try It Yourself

> Tap **Try It Yourself**, play with the codes, create your own interesting examples, and share them in the comments below.

Array Destructuring in ES6

The **destructuring** assignment syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.
ES6 has added a shorthand syntax for destructuring an array.

The following example demonstrates how to unpack the elements of an array into distinct variables:

```
let arr = ['1', '2', '3'];
let [one, two, three] = arr;

console.log(one); // 1
console.log(two); // 2
console.log(three); // 3
```

Try It Yourself

We can use also destructure an array returned by a function.
For example:

```
let a = () => {
  return [1, 3, 2];
};

let [one, , two] = a();
```

Try It Yourself

Notice that we left the second argument's place empty.

The destructuring syntax also simplifies assignment and swapping values:

```
let a, b, c = 4, d = 8;
[a, b = 6] = [2]; // a = 2, b = 6

[c, d] = [d, c]; // c = 8, d = 4
```

Try It Yourself

> Tap **Try it Yourself** to play around with the code.

Object Destructuring in ES6

Similar to Array destructuring, **Object destructuring** unpacks properties into distinct variables.
For example:

```
let obj = {h:100, s: true};
let {h, s} = obj;

console.log(h); // 100
console.log(s); // true
```

Try It Yourself

We can assign without declaration, but there are some syntax requirements for that:

```
let a, b;
({a, b} = {a: 'Hello ', b: 'Jack'});

console.log(a + b); // Hello Jack
```

Try It Yourself

The () with a **semicolon** (;) at the end are **mandatory** when destructuring without a declaration. However, you can also do it as follows where the () are not required:

```
let {a, b} = {a: 'Hello ', b: 'Jack'};
console.log(a + b);
```

Try It Yourself

You can also assign the object to new variable names.
For example:

```
var o = {h: 42, s: true};
var {h: foo, s: bar} = o;

//console.log(h); // Error
console.log(foo); // 42
```

Try It Yourself

Finally you can assign **default values** to variables, in case the value unpacked from the object is undefined.
For example:

```
var obj = {id: 42, name: "Jack"};

let {id = 10, age = 20} = obj;

console.log(id); // 42
console.log(age); // 20
```

Try It Yourself

Tap **Try it Yourself** to play around with the code.

ES6 Rest Parameters

Prior to ES6, if we wanted to pass a variable number of arguments to a function, we could use the **arguments** object, an array-like object, to access the parameters passed to the function.
For example, let's write a function that checks if an array contains all the arguments passed:

```
function containsAll(arr) {
  for (let k = 1; k < arguments.length; k++) {
    let num = arguments[k];
    if (arr.indexOf(num) === -1) {
      return false;
    }
```

```
  }
  return true;
}
let x = [2, 4, 6, 7];
console.log(containsAll(x, 2, 4, 7));
console.log(containsAll(x, 6, 4, 9));
```

Try It Yourself

We can pass any number of arguments to the function and access it using the **arguments** object.

While this does the job, ES6 provides a more readable syntax to achieve variable number of parameters by using a **rest parameter**:

```
function containsAll(arr, ...nums) {
  for (let num of nums) {
    if (arr.indexOf(num) === -1) {
      return false;
    }
  }
  return true;
}
```

Try It Yourself

The **...nums** parameter is called a **rest parameter**. It takes all the "extra" arguments passed to the function. The three dots (...) are called the **Spread operator**.

> Only the last parameter of a function may be marked as a rest parameter. If there are no extra arguments, the rest parameter will simply be an empty array; the rest parameter will never be **undefined**.

The Spread Operator

This operator is similar to the Rest Parameter, but it has another purpose when used in objects or arrays or function calls (arguments).

Spread in function calls

It is common to pass the elements of an array as arguments to a function. Before ES6, we used the following method:

```
function myFunction(w, x, y, z) {
  console.log(w + x + y + z);
}
var args = [1, 2, 3];
myFunction.apply(null, args.concat(4));
```

Try It Yourself

ES6 provides an easy way to do the example above with **spread operators**:

```
const myFunction = (w, x, y, z) => {
  console.log(w + x + y + z);
};
let args = [1, 2, 3];
myFunction(...args, 4);
```

Try It Yourself

Example:

```
var dateFields = [1970, 0, 1];  // 1 Jan 1970
var date = new Date(...dateFields);
console.log(date);
```

Try It Yourself

Spread in array literals

Before ES6, we used the following syntax to add an item at middle of an array:

```
var arr = ["One", "Two", "Five"];

arr.splice(2, 0, "Three");
arr.splice(3, 0, "Four");
console.log(arr);
```

Try It Yourself

You can use methods such as push, splice, and concat, for example, to achieve this in different positions of the array. However, in ES6 the spread operator lets us do this more easily:

```
let newArr = ['Three', 'Four'];
let arr = ['One', 'Two', ...newArr, 'Five'];
console.log(arr);
```

Try It Yourself

Spread in object literals

In objects it copies the own enumerable properties from the provided object onto a new object.

```
const obj1 = { foo: 'bar', x: 42 };
const obj2 = { foo: 'baz', y: 5 };

const clonedObj = {...obj1}; // { foo: "bar", x: 42 }
const mergedObj = {...obj1, ...obj2}; // { foo: "baz", x: 42, y: 5 }
```

Try It Yourself

However, if you try to merge them you will not get the result you expected:

```
const obj1 = { foo: 'bar', x: 42 };
const obj2 = { foo: 'baz', y: 5 };
const merge = (...objects) => ({...objects});

let mergedObj = merge(obj1, obj2);
// { 0: { foo: 'bar', x: 42 }, 1: { foo: 'baz', y: 5 } }

let mergedObj2 = merge({}, obj1, obj2);
// { 0: {}, 1: { foo: 'bar', x: 42 }, 2: { foo: 'baz', y: 5 } }
```

Try It Yourself

> Shallow cloning or merging objects is possible with another operator called **Object.assign()**.

Classes in ES6

In this lesson we'll explain how to create a **class** that can be used to create multiple objects of the same structure.
A class uses the keyword **class** and contains a **constructor** method for initializing.

For example:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

A declared class can then be used to create multiple objects using the keyword **new**.
For example:

```
const square = new Rectangle(5, 5);
const poster = new Rectangle(2, 3);
```

Try It Yourself

> Class Declarations are **not hoisted** while Function Declarations are. If you try to access your class before declaring it, **ReferenceError** will be returned.

You can also define a class with a **class expression**, where the class can be named or unnamed.
A **named** class looks like:

```
var Square = class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Try It Yourself

In the unnamed class expression, a variable is simply assigned the class definition:

```
var Square = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Try It Yourself

> The **constructor** is a special method which is used for creating and initializing an object created with a class.
> There can be only **one** constructor in each class.

Class Methods in ES6

ES6 introduced a shorthand that does not require the keyword **function** for a function assigned to a method's name. One type of class method is the **prototype method**, which is available to objects of the class.
For Example:

```
class Rectangle {
 constructor(height, width) {
   this.height = height;
   this.width = width;
 }
 get area() {
   return this.calcArea();
 }
 calcArea() {
   return this.height * this.width;
 }
}
const square = new Rectangle(5, 5);
console.log(square.area); // 25
```

Try It Yourself

In the code above, **area** is a getter, **calcArea** is a method.

Another type of method is the **static method**, which cannot be called through a class instance. Static methods are often used to create utility functions for an application.
For Example:

```
class Point {
 constructor(x, y) {
   this.x = x;
   this.y = y;
 }
 static distance(a, b) {
   const dx = a.x - b.x;
   const dy = a.y - b.y;
   return Math.hypot(dx, dy);
 }
}
const p1 = new Point(7, 2);
const p2 = new Point(3, 8);

console.log(Point.distance(p1, p2));
```

Try It Yourself

As you can see, the static **distance** method is called directly using the class name, without an object.

Inheritance in ES6

The **extends** keyword is used in class declarations or class expressions to create a child of a class. The child inherits the properties and methods of the parent.
For example:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}

let dog = new Dog('Rex');
dog.speak();
```

Try It Yourself

In the code above, the **Dog** class is a child of the **Animal** class, inheriting its properties and methods.

> If there is a constructor present in the subclass, it needs to first call **super**() before using **this**. Also, the **super** keyword is used to call parent's methods.

For example, we can modify the program above to the following:

```
class Animal {
  constructor(name) {
    this.name = name;
  }
  speak() {
    console.log(this.name + ' makes a noise.');
  }
}
}

class Dog extends Animal {
  speak() {
    super.speak(); // Super
    console.log(this.name + ' barks.');
  }
}

let dog = new Dog('Rex');
dog.speak();
```

Try It Yourself

> In the code above, the parent's **speak()** method is called using the **super** keyword.

ES6 Map

A **Map** object can be used to hold **key/value** pairs. A key or value in a map can be anything (objects and primitive values).

The syntax **new Map([iterable])** creates a Map object where **iterable** is an array or any other iterable object whose elements are arrays (with a key/value pair each).

An **Object** is similar to **Map** but there are important differences that make using a Map preferable in certain cases:
1) The keys can be any type including functions, objects, and any primitive.
2) You can get the size of a Map.
3) You can directly iterate over Map.
4) Performance of the Map is better in scenarios involving frequent addition and removal of key/value pairs.

The **size** property returns the number of key/value pairs in a map.
For example:

```
let map = new Map([['k1', 'v1'], ['k2', 'v2']]);

console.log(map.size); // 2
```

Try It Yourself

Methods
set(key, value) Adds a specified key/value pair to the map. If the specified key already exists, value corresponding to it is replaced with the specified value.
get(key) Gets the value corresponding to a specified key in the map. If the specified key doesn't exist, undefined is returned.
has(key) Returns true if a specified key exists in the map and false otherwise.
delete(key) Deletes the key/value pair with a specified key from the map and returns true. Returns false if the element does not exist.
clear() Removes all key/value pairs from map.
keys() Returns an Iterator of keys in the map for each element.
values() Returns an Iterator of values in the map for each element.
entries() Returns an Iterator of array[key, value] in the map for each element.

For example:

```
let map = new Map();

map.set('k1', 'v1').set('k2', 'v2');

console.log(map.get('k1')); // v1

console.log(map.has('k2')); // true

for (let kv of map.entries())
  console.log(kv[0] + " : " + kv[1]);
```

Try It Yourself

The above example demonstrates some of the ES6 Map methods.

> **Map** supports different data types i.e. 1 and "1" are two different keys/values.

ES6 Set

A **Set** object can be used to hold **unique** values (no repetitions are allowed).
A value in a set can be anything (objects and primitive values).

The syntax **new Set([iterable])** creates a Set object where **iterable** is an array or any other iterable object of values.

The **size** property returns the number of distinct values in a set.
For example:

```
let set = new Set([1, 2, 4, 2, 59, 9, 4, 9, 1]);

console.log(set.size); // 5
```

Try It Yourself

Methods
add(value) Adds a new element with the given value to the Set.
delete(value) Deletes a specified value from the set.
has(value) Returns true if a specified value exists in the set and false otherwise.
clear() Clears the set.
values() Returns an Iterator of values in the set.

For example:

```
let set = new Set();

set.add(5).add(9).add(59).add(9);

console.log(set.has(9));

for (let v of set.values())
  console.log(v);
```

Try It Yourself

The above example demonstrates some of the ES6 Set methods.

> **Set** supports different data types i.e. **1** and **"1"** are two different values.
> NaN and undefined can also be stored in Set.

ES6 Promises

A **Promise** is a better way for asynchronous programming when compared to the common way of using a **setTimeout**() type of method.

Consider this **example:**

```
setTimeout(function() {
  console.log("Work 1");
  setTimeout(function() {
    console.log("Work 2");
  }, 1000);
}, 1000);
console.log("End");
```

Try It Yourself

It prints "End", "Work 1" and "Work 2" in that order (the work is done asynchronously). But if there are more events like this, the code becomes very complex.

ES6 comes to the rescue in such situations. A **promise** can be created as follows:

```
new Promise(function(resolve, reject) {
   // Work
   if (success)
      resolve(result);
   else
      reject(Error("failure"));
});
```

Here, **resolve** is the method for success and **reject** is the method for failure.

If a method returns a promise, its calls should use the **then** method which takes two methods as input; one for success and the other for the failure.

For Example:

```
function asyncFunc(work) {
  return new Promise(function(resolve, reject) {
    if (work === "")
      reject(Error("Nothing"));
    setTimeout(function() {
      resolve(work);
    }, 1000);
  });
}

asyncFunc("Work 1") // Task 1
.then(function(result) {
  console.log(result);
  return asyncFunc("Work 2"); // Task 2
}, function(error) {
  console.log(error);
})
.then(function(result) {
  console.log(result);
}, function(error) {
  console.log(error);
});
console.log("End");
```

Try It Yourself

It also prints "End", "Work 1" and "Work 2" (the work is done asynchronously). But, this is clearly more readable than the previous example and in more complex situations it is easier to work with.

> Tap **Try It Yourself** to play around with the codes and see ES6 Promises in action.

Iterators & Generators

Symbol.iterator is the default iterator for an object. The for...of loops are based on this type of iterator.

In the example below, we will see how we should implement it and how **generator functions** are used.
Example:

```
let myIterableObj = {
  [Symbol.iterator] : function* () {
    yield 1; yield 2; yield 3;
...
console.log([...myIterableObj]);
```

Try It Yourself

First, we create an object, and use the **Symbol.iterator** and **generator function** to fill it with some values.

In the second line of the code, we use a * with the **function** keyword. It's called a **generator function (or gen function)**.

For example, here is a simple case of how **gen functions** can be useful:

```
function* idMaker() {
  let index = 0;
  while (index < 5)
    yield index++;
}
var gen = idMaker();
console.log(gen.next().value);
```

Try It Yourself

We can exit and re-enter generator functions later. Their variable bindings (context) will be saved across re-entrances. They are a very powerful tool for asynchronous programming, especially when combined with Promises. They can also be useful for creating loops with special requirements.

We can nest **generator functions** inside each other to create more complex structures and pass them arguments while we are calling them.
The example below will show a useful case of how we can use **generator functions** and **Symbol.iterators** together.
Example:

```
const arr = ['0', '1', '4', 'a', '9', 'c', '16'];
const my_obj = {
  [Symbol.iterator]: function*() {
    for(let index of arr) {
      yield `${index}`;
    }
  }
};
const all = [...my_obj]
  .map(i => parseInt(i, 10))
  .map(Math.sqrt)
  .filter((i) => i < 5)
  .reduce((i, d) => i + d);
console.log(all);
```

Try It Yourself

We create an object of 7 elements by using **Symbol.iterator** and **generator functions**. In the second part, we assign our object to a constant **all**. At the end, we print its value.

> Tap **Try It Yourself** and follow the instructions in the comments to see the results.

Modules

It is a good practice to divide your related code into modules. Before ES6 there were some libraries which made this possible (e.g., RequireJS, CommonJS). ES6 is now supporting this feature natively.

Considerations when using modules:
The first consideration is **maintainability**. A module is independent of other modules, making improvements and expansion possible without any dependency on code in other modules.

The second consideration is **namespacing**. In an earlier lesson, we talked about variables and scope. As you know, vars are globally declared, so it's common to have namespace pollution where unrelated variables are accessible all over our code. Modules solve this problem by creating a private space for variables.

Another important consideration is **reusability**. When we write code that can be used in other projects, modules make it possible to easily reuse the code without having to rewrite it in a new project.

Let's see how we should use modules in JS files.
For Example:

```
// lib/math.js
export let sum = (x, y) => { return x + y; }
export let pi = 3.14;

// app.js
import * as math from "lib/math"
console.log(`2p = + ${math.sum(math.pi, math.pi)}`)
```

Here we are exporting the sum function and the pi variable so we can use them in different files.

> ES6 supports **modules** officially, however, some browsers are not supporting modules natively yet. So, we should use bundlers (builders) such as Webpack or Browserify to run our code.

Built-in Methods

ES6 also introduced new built-in methods to make several tasks easier. Here we will cover the most common ones.

Array Element Finding

The legacy way to find the first element of an array by its value and a rule was the following:

```
[4, 5, 1, 8, 2, 0].filter(function (x) {
  return x > 3;
})[0];
```

Try It Yourself

The new syntax is cleaner and more robust:

```
[4, 5, 1, 8, 2, 0].find(x => x > 3);
```

Try It Yourself

You can also get the index of the item above by using the **findIndex()** method:

```
[4, 5, 1, 8, 2, 0].findIndex(x => x > 3);
```

Try It Yourself

Repeating Strings

Before ES6 the following syntax was the correct way to repeat a string multiple times:

```
console.log(Array(3 + 1).join("foo")); // foofoofoo
```

Try It Yourself

With the new syntax, it becomes:

```
console.log("foo".repeat(3)); // foofoofoo
```

Try It Yourself

Searching Strings

Before ES6 we only used the indexOf() method to find the position of the text in the string. For example:

```
"SoloLearn".indexOf("Solo") === 0; // true
"SoloLearn".indexOf("Solo") === (4 - "Solo".length); // true
"SoloLearn".indexOf("loLe") !== -1; // true
"SoloLearn".indexOf("olo", 1) !== -1; // true
"SoloLearn".indexOf("olo", 2) !== -1; // false
```

Try It Yourself

ES6 has replaced this with a version that has cleaner and more simplified syntax:

```
"SoloLearn".startsWith("Solo", 0); // true
"SoloLearn".endsWith("Solo", 4); // true
"SoloLearn".includes("loLe"); // true
"SoloLearn".includes("olo", 1); // true
"SoloLearn".includes("olo", 2); // false
```

Try It Yourself

It is always a good practice to refactor your code with the new syntax to learn new things and make your code more understandable.

End.