ES6 & JS Design Patterns

Object Oriented JavaScript

ITI – Assiut Branch
Eng. Hany Saad

Closures

Closures

- Closure is one of the most powerful features of JavaScript.
- A closure is an expression (typically a function) that can have free variables together with an environment that binds those variables (that "closes" the expression).
- It is created when the inner function is somehow made available to any scope outside the outer function.
- If the inner function manages to survive beyond the life of the outer function; the variables and functions defined in the outer function will live longer than the outer function itself, since the inner function has access to the scope of the outer function.
- In short words:
  - a closure is the local variables for a function — kept alive after the function has returned

Closures (cont.)

- **Example:**

```
function sayHello2(name) {
    var text = 'Hello ' + name; // Local variable
    var sayAlert = function() { alert(text); }
    return sayAlert;    //returning reverence to the inner func.
}
var say2 = sayHello2('Bob');
//say2 holds a reference to the inner func. That access the outer func variables.
say2(); // alerts "Hello Bob"
```

- The above code has a closure because the anonymous function function() { alert(text); } is declared inside another function, sayHello2() in this example. In JavaScript, if you use the function keyword inside another function, you are creating a closure.

- In JavaScript, if you declare a function within another function, then the local variables can remain accessible after returning from the function you called. This is demonstrated above, because we call the function say2() after we have returned from sayHello2(). Notice that the code that we call access the variable text, which was a local variable of the function sayHello2().

- The anonymous function can reference text which holds the value 'Hello Bob' because the local variables of sayHello2() are kept in a closure.

- The magic is that in JavaScript a function reference also has a secret reference to the closure it was created in.

Closures (cont.)

- **Another Example (Problem):**

```
function closureTest(){
    var arr = [];
    for(var i = 0; i < 3; i ++) {
        arr.push(function(){
                    alert(i);
             });
    }
    return arr;
}
var cFn = closureTest();
cFn[0](); //3
cFn[1](); //3
cFn[2](); //3
```

- Note that when you run the example, “3" is alerted three times! This is because there is only one closure for the local variables for closureTest.
- When the anonymous functions are called on the line cFn[0](); they all use the same single closure, and they use the current value for i and item within that one closure (where i has a value of 3 because the loop had completed, and item has a value of ‘3').

Closures (cont.)

- ## Another Example (Solution):

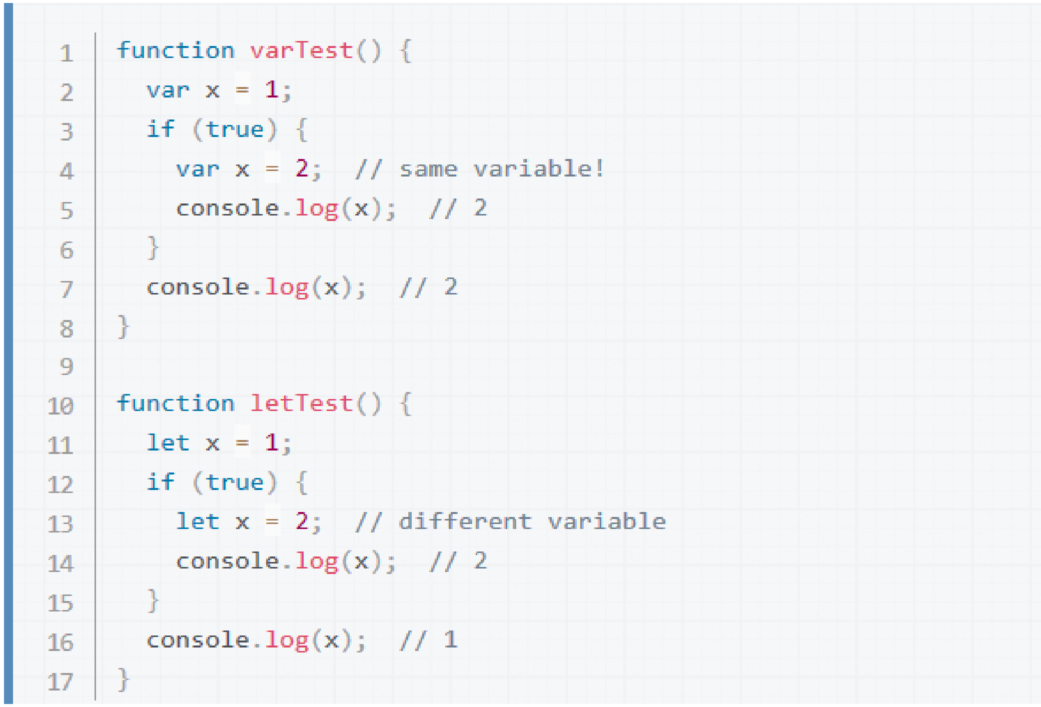
```
function closureTest(){
        var arr=[]
        var i;
        for(var i = 0; i < 3; i ++){
        arr.push((function(j){ return function(){
                                         alert(j);
                                 }
                         })(i)
                );
        }
 return arr;
}
var cFn = closureTest();
cFn[0](); //0
cFn[1](); //1
cFn[2](); //2
```

ES6 new features

Variables – block scope with let

❑ Block variable declaration: let (New ES6 feature):

- There was no Block Scope before ES6, only function scope, let declaration introduced in ES6 allowing block scope
- Variables declared by let have as their scope the block in which they are defined, as well as in any contained sub-blocks .
- let variables are block-scoped. The scope of a variable declared with let is just the enclosing block, not the whole enclosing function.



```
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
}
```

Variables – block scope with let (Cont.)

❑ Block variable declaration: let (Cont.):

- Loops of the form for (let x...) create a fresh binding for x in each iteration, and the scope of the variable will be inside the for loop only.

```
function test(){
        ……
        for (let i = 0; i < messages.length; i++) {
                ... //let scope inside loop only, not whole function.
        }
}
```

- Global let variables are not properties on the global object. That is, you won’t access them by writing window.variableName. Instead, they live in the scope of an invisible block that notionally encloses all JS code that runs in a web page.
- It’s an error to try to use a let variable before its declaration is reached (as variables declared using let aren’t hoisted).

```
function update() {
  document.write("your name:", t);  // ReferenceError
  ...
  let t = "test";}
```

Variables - Constants

❑ JavaScript Constants (new ES6 Feature):

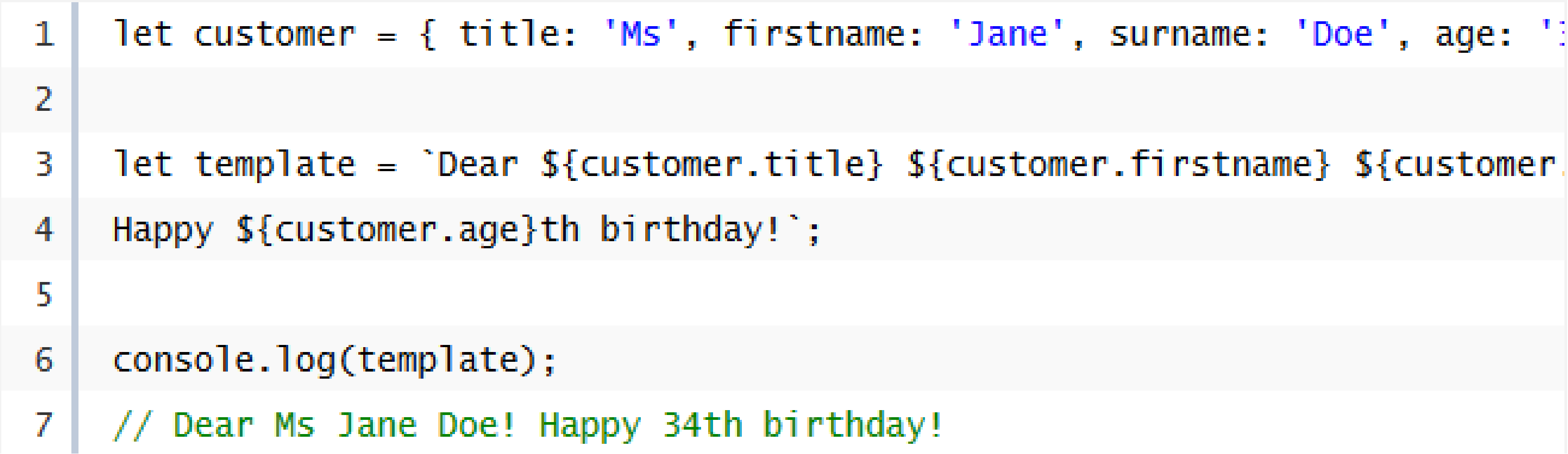
- Variables declared with const are constant variables, youcan’t assign to them, except at the point where they’re declared.

```
const MAX_CAT_SIZE_KG = 3000;
MAX_CAT_SIZE_KG = 5000; // SyntaxError
MAX_CAT_SIZE_KG++; // SyntaxError
const theFairest;  // SyntaxError, you can’t declare const variable without assigning it a value
```

- A constant can be global or local to a function where it is declared.
- Constants also share a feature with variables declared using let in that they are block-scoped instead of function-scoped (and thus they are not hoisted)

Template Literals

- Template literals allow us to easily create templates in which we can embed different values to any spot we want.
- To do so we need to use the ${...} syntax everywhere where we want to insert the data that we can pass in from variables, arrays, or objects.

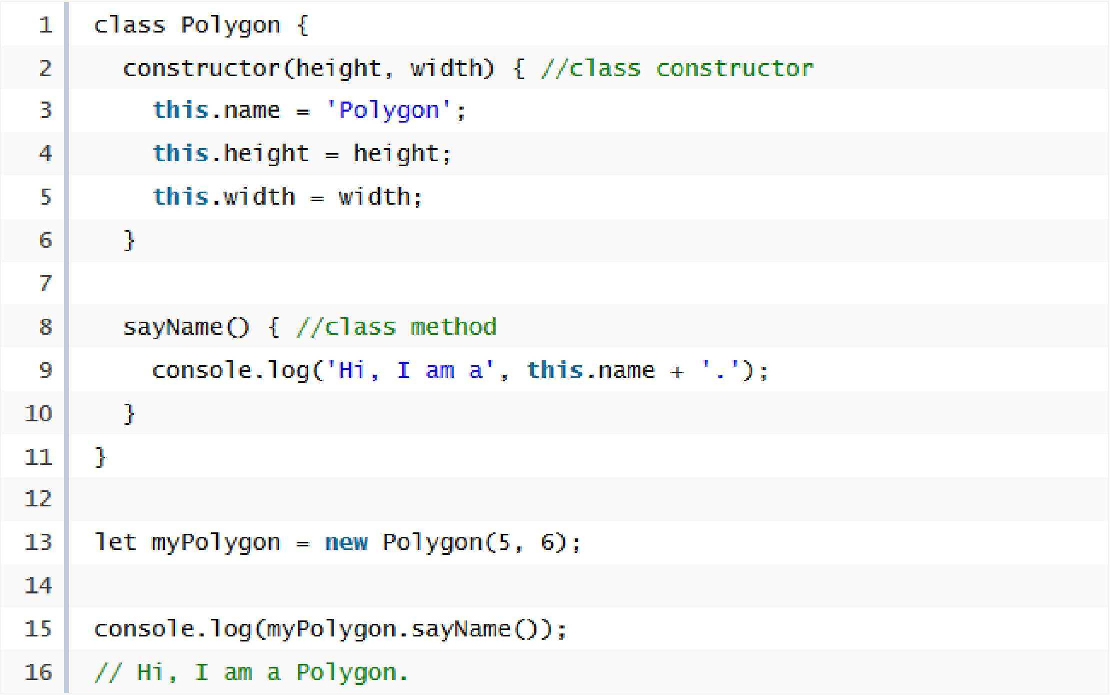


```
let customer = { title: 'Ms', firstname: 'Jane', surname: 'Doe', age: '

let template = `Dear ${customer.title} ${customer.firstname} ${customer
Happy ${customer.age}th birthday!`;

console.log(template);
// Dear Ms Jane Doe! Happy 34th birthday!
```

Demo!

Classes

- ES6 introduces JavaScript classes that are built upon the existing prototype-based inheritance.
- The new syntax makes it more straightforward to create objects, take leverage of inheritance, and reuse code.
- Classes are in fact "special functions", and just as you can define function expressions and function declarations, the class syntax has two components: class expressions and class declarations.



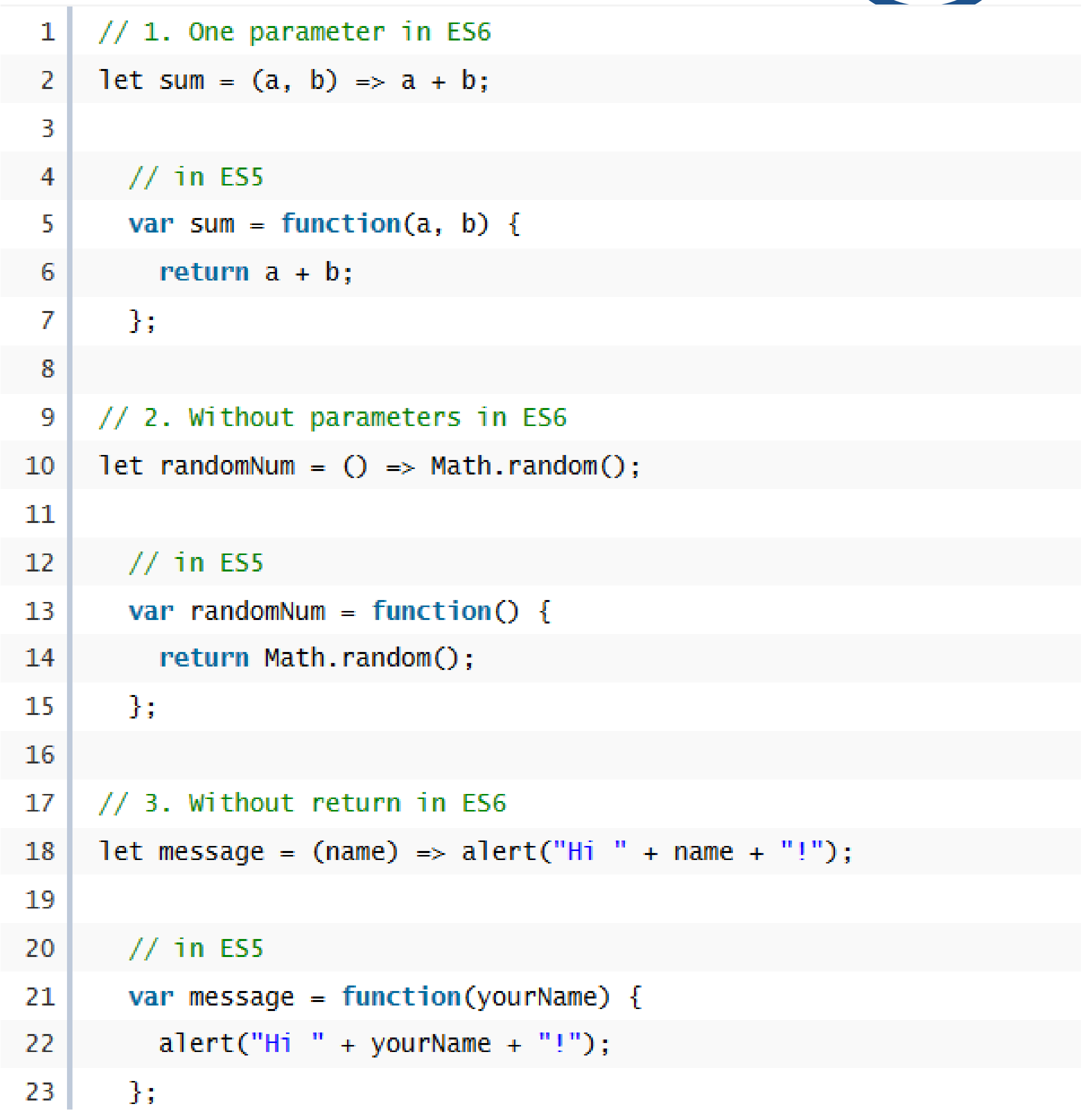
```
class Polygon {
  constructor(height, width) { //class constructor
    this.name = 'Polygon';
    this.height = height;
    this.width = width;
  }

  sayName() { //class method
    console.log('Hi, I am a', this.name + '.');
  }
}

let myPolygon = new Polygon(5, 6);

console.log(myPolygon.sayName());
// Hi, I am a Polygon.
```

Demo!

Arrow functions

- ECMAScript 6 facilitates how we write anonymous functions, as we can completely omit the function keyword.

- We only need to use the new syntax for arrow functions, named after the => arrow sign (fat arrow), that provides us with a great shortcut.



```
// 1. One parameter in ES6
let sum = (a, b) => a + b;

  // in ES5
  var sum = function(a, b) {
    return a + b;
  };

// 2. Without parameters in ES6
let randomNum = () => Math.random();

  // in ES5
  var randomNum = function() {
    return Math.random();
  };

// 3. Without return in ES6
let message = (name) => alert("Hi " + name + "!");

  // in ES5
  var message = function(yourName) {
    alert("Hi " + yourName + "!");
  };
```

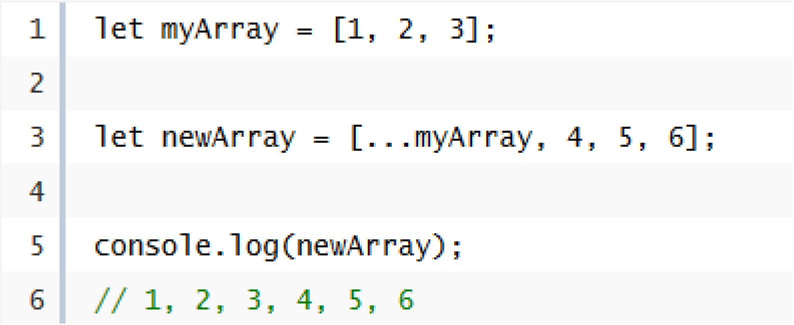
Arrow functions

- Before arrow functions, every new function defined its own this value (a new object in the case of a constructor, undefined in strict mode function calls, the context object if the function is called as an "object method", etc.).
- An arrow function does not create its own this context, so this has its original meaning from the enclosing context.

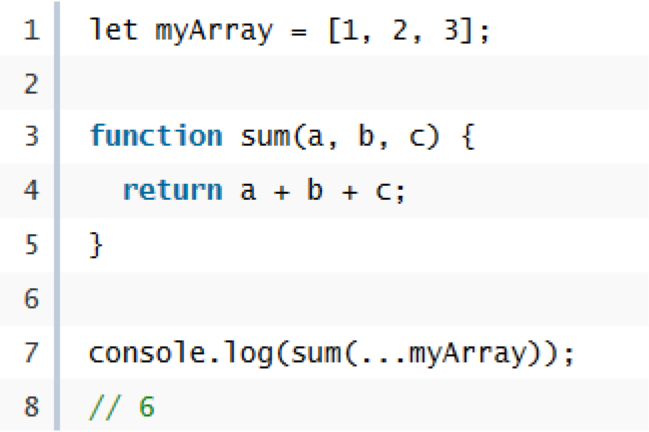
Demo!

New spread Operator

- The new spread operator is marked with 3 dots (…), and we can use it to sign the place of multiple expected items.
- One of the most common use cases of the spread operator is:
  - inserting the elements of an array into another array.
  - We can also take leverage of the spread operator in function calls in which we want to pass in arguments from an array.



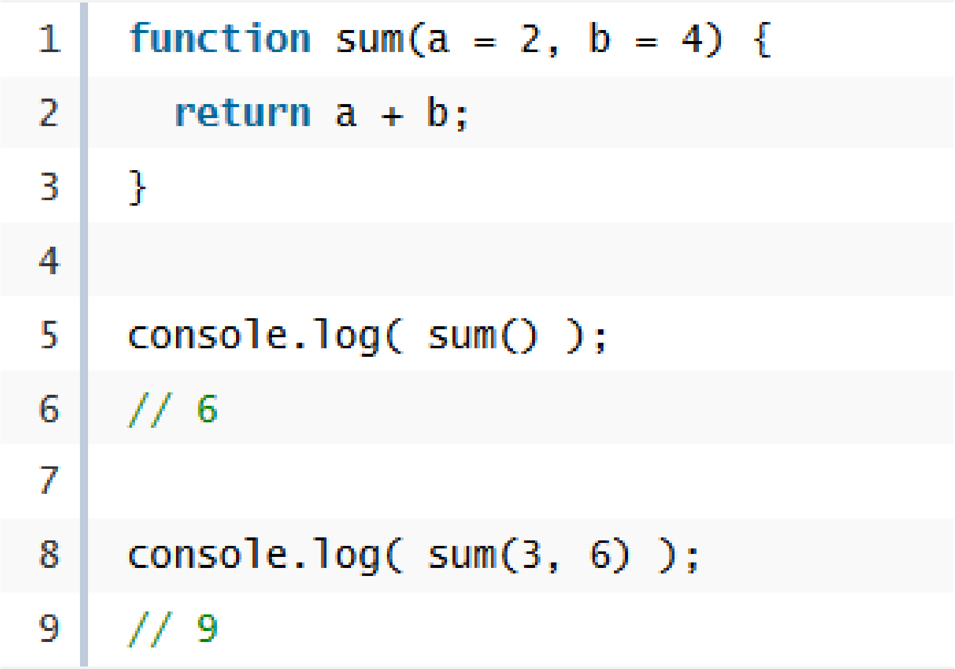
```
let myArray = [1, 2, 3];

let newArray = [...myArray, 4, 5, 6];

console.log(newArray);
// 1, 2, 3, 4, 5, 6
```



```
let myArray = [1, 2, 3];

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

console.log(sum(...myArray));
// 6
```

Default Values for Parameters & New Rest Parameters

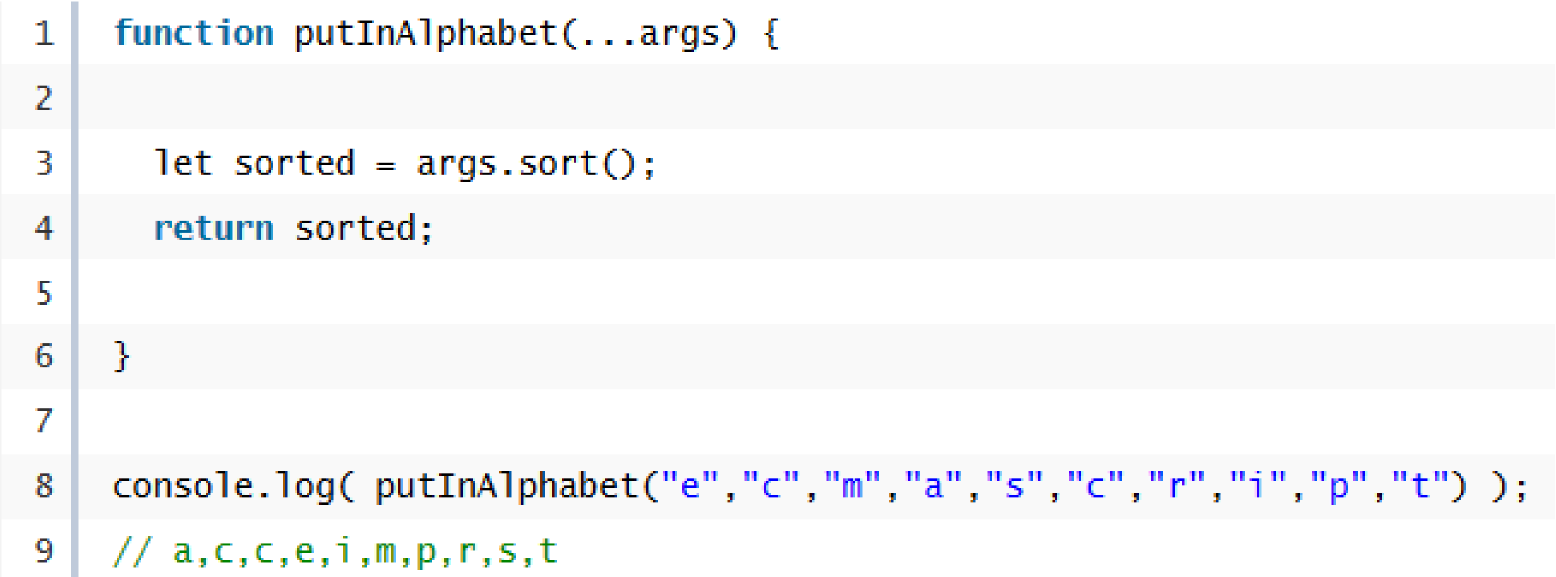
- In ES5 the default values of parameters are always set to undefined.
- In ECMAScript 6 we can add default values to the parameters of a function.



```
function sum(a = 2, b = 4) {
  return a + b;
}

console.log( sum() );
// 6

console.log( sum(3, 6) );
// 9
```

Default Values for Parameters & New Rest Parameters (Cont.)

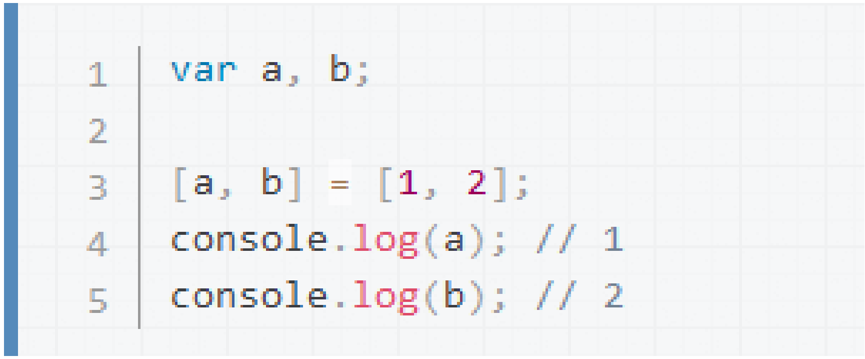
- ES6 also introduces a new kind of parameter, the rest parameters.
- They look and work similarly to spread operators, They come handy if we don’t know how many arguments will be passed in later in the function call.



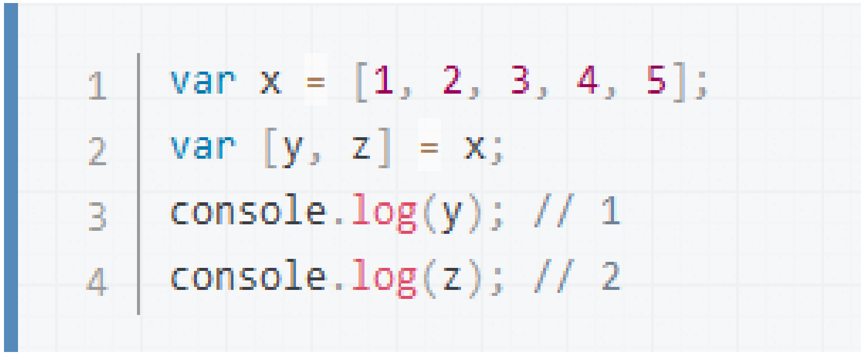
```
function putInAlphabet(...args) {

  let sorted = args.sort();
  return sorted;

}

console.log( putInAlphabet("e","c","m","a","s","c","r","i","p","t") );
// a,c,c,e,i,m,p,r,s,t
```

Destructuring assignment

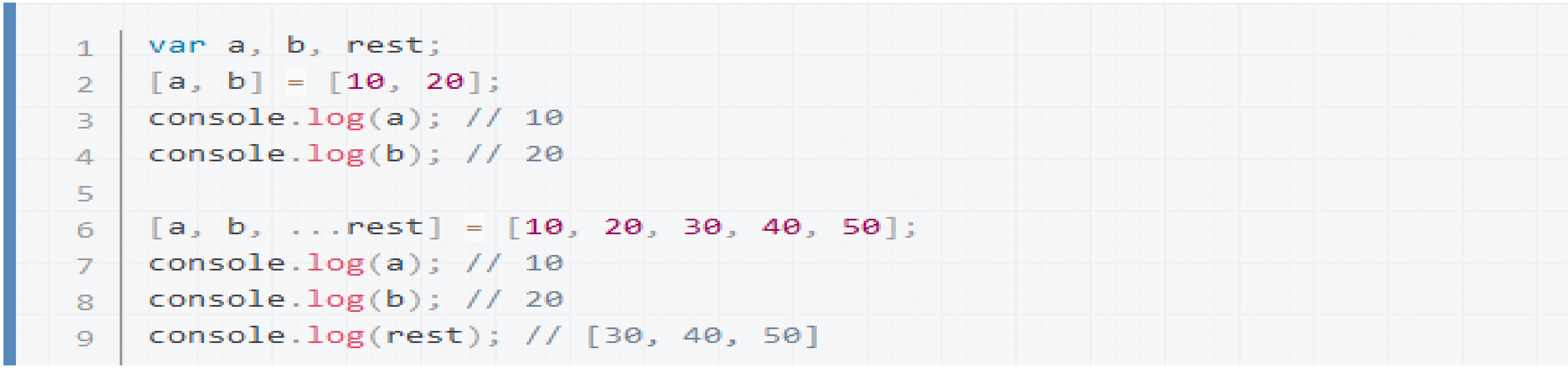
- The destructuring assignment syntax is a JavaScript expression that makes it possible to extract data from arrays or objects into distinct variables.



```
var a, b;

[a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2
```



```
var x = [1, 2, 3, 4, 5];
var [y, z] = x;
console.log(y); // 1
console.log(z); // 2
```



```
var a, b, rest;
[a, b] = [10, 20];
console.log(a); // 10
console.log(b); // 20

[a, b, ...rest] = [10, 20, 30, 40, 50];
console.log(a); // 10
console.log(b); // 20
console.log(rest); // [30, 40, 50]
```

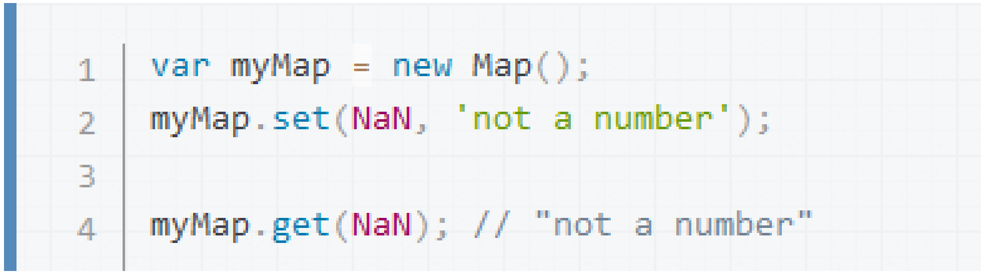
Sets

- The Set object lets you store unique values of any type, whether primitive values or object references.
- Syntax: **var mySet=new Set([iterable]);**
  - If an iterable object is passed, all of its elements will be added to the new Set. If null is passed instead of iterable, it is treated as not passing iterable at all.
- More details:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set

Demo!

Maps

- The Map object is a simple key/value map. Any value (both objects and primitive values) may be used as either a key or a value.
- Syntax: **var new Map([iterable]);**
  - Iterable is an Array or other iterable object whose elements are key-value pairs (2-element Arrays). Each key-value pair is added to the new Map. null is treated as undefined.
- Maps Vs. Objects:
  - Map instances are only useful for collections, and you should consider adapting your code where you have previously used objects for such.
  - Objects shall be used as records, with fields and methods.
  - If you're still not sure which one to use
- More details:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map



```
var myMap = new Map();
myMap.set(NaN, 'not a number');

myMap.get(NaN); // "not a number"
```

Demo!

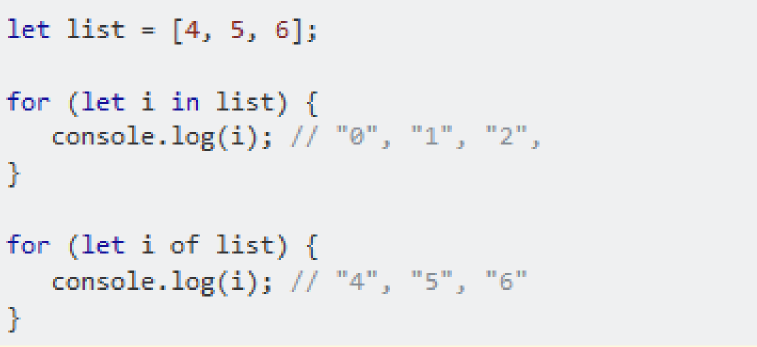
Generators

- Generators are functions which can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances.
- Calling a generator function does not execute its body immediately; an iterator object for the function is returned instead.
  - When the iterator's next() method is called, the generator function's body is executed until the first yield expression , which specifies the value to be returned from the iterator or, with yield*, delegates to another generator function.
  - The next() method returns an object with a value property containing the yielded value and a done property which indicates whether the generator has yielded its last value as a boolean.
- The function* declaration (function keyword followed by an asterisk) defines a generator function, which returns a Generator object.

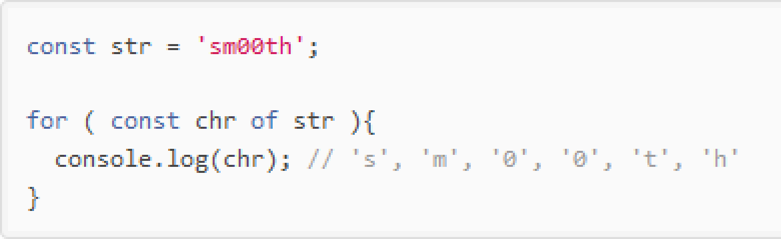
Demo!

for..of

- The famous for..in loop whose first value is to iterate over the different keys of an object or an array.
  - When itarating over an array, index value is parsed to string : "0", "1", "2", etc.. This behaviour can lead to potential error when index is used in computation.
- The alternative .forEach() method oop allow a more secure iteration, but bring other downsides as:
  - Impossibility to halt the loop with the traditional break; and return; statements.
  - Array only dedicated method.
- ECMA consortium has so decided to proceed with establishment of a new enhanced version of the for..in loop. Thus was born the for..of loop which, from now on, will coexist with the previous one allowing to maintain the backward compatibility with former version of the standard.
  - for–of is not just for arrays. It also works on most array-like objects, like DOM NodeLists.
  - It also works on strings, treating the string as a sequence of Unicode characters.



```
let list = [4, 5, 6];

for (let i in list) {
   console.log(i); // "0", "1", "2",
}

for (let i of list) {
   console.log(i); // "4", "5", "6"
}
```



```
const str = 'sm00th';

for ( const chr of str ){
  console.log(chr); // 's', 'm', '0', '0', 't', 'h'
}
```

for..of (Cont.)

- In a nutshel for..of comes to:
  - Address for..in loop gaps
  - Allow a simplified iteration over iterable objects (Array, String, Maps, Sets, Generators,NodeList, arguments)
  - Unlike .foreach()Allow using break, continue, return.

Modules

- Modules are one of the most important features of any programming language.
- Sadly, JavaScript lacks this very basic feature. But, that doesn’t stop us from writing modular code. We have two important standards, namely CommonJS and Asynchronous Module Definition (AMD) which let developers use modules in JavaScript. But, the next JavaScript version, known as ECMAScript 6, brings modules into JavaScript officially.
- In ES6 each module is defined in its own file. The functions or variables defined in a module are not visible outside unless you explicitly export them. This means that you can write code in your module and only export those values which should be accessed by other parts of your app.
- To export certain variables from a module you just use the keyword export. Similarly, to consume the exported variables in a different module you use import.

Demo!

Resources

- **Online Resources:**
  - http://www.hongkiat.com/blog/ecmascript-6/
  - https://webapplog.com/es6/
  - http://exploringjs.com/es6/ch_overviews.html
  - https://developer.mozilla.org
  - https://developers.google.com/web/fundamentals/getting-started/primers/promises
  - https://leanpub.com/understandinges6/read/
- **Books:**
  - *Understanding ECMAScript 6* by Nicolas Zakas book
  - ES6 Cheatsheet (FREE PDF)
  - *Exploring ES6* by Dr. Axel Rauschmayer

<SCRIPT >

</SCRIPT>

```
<script>document.writeln(“Thank You!”)</script>
```