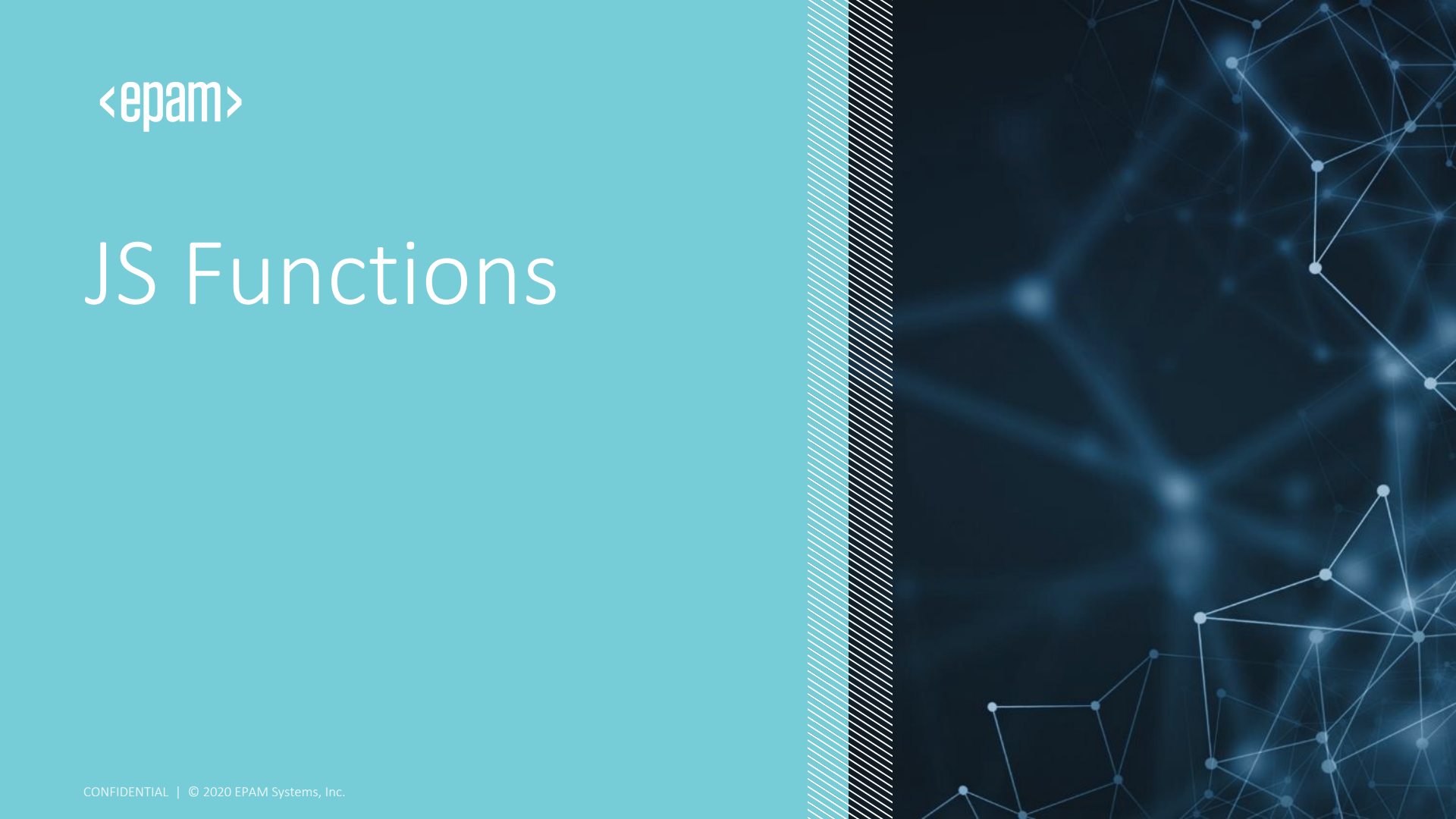


JS Functions

AGENDA

1. FUNCTION DECLARATION
2. FUNCTION EXPRESSION
3. ARROW FUNCTIONS
4. SETTIMEOUT AND SETINTERVAL
5. THIS
6. PROMISES. ASYNC/AWAIT

definition

Function is a block of code

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition

Functions should do only one "function"

In ECMAScript, functions are the *first-class* objects. This term means that functions may be passed as arguments to other functions

JS functions can accept arguments, return a value or simply skip any of them

```
function meow() {
 // function body
 console.log('meow');
}

// Call function
meow();

function getName() {
 // function body
 return 'Petro';
}

let name = getName();
console.log(name);
```

Function Features

Function is an object. But also it can be invoked.

Function can have name or be anonymous

Function has a body

Function object has some predefined properties and methods.

```
function outerFunction() {  // The most common declaration
 // everything inside a { } brackets is a function body
 var innerFunction = function() { // Functional expression, function is anonymous
    // here is a body of inner function;
 }
}

outerFunction(); // we use () to invoke a function

console.log(typeof outerFunction); // "function" - skip () to work with object
console.log(outerFunction.name); // -> "outerFunction"
console.log(outerFunction.toString()); // -> text of a function body
```

Function is an Object

We can save function to variable

```
let myFunctionSaved = function myFunction() {...}
```

We can pass it as a parameter to another function

```
function a() {console.log("a was called");}
function b(incomingFunction) {
 incomingFunction();
}
// call function "b" and pass function "a" as an object to params
b(a); // -> "a was called";
```

We can return it from another function

```
function b() {
 return function a () {
  console.log(a.name + " was called");
 }
}
var returnedFunction = b(); // call function "b" and receive a function object
returnedFunction(); // -> "a was called";
```

Function Scope Basics

1. Scope is a hidden object, no direct access
2. Scope is a storage for all declared variables and functions
3. Function scope is created on each function call.
4. Function parameters become scope variables

```
function writeToScope(a, b) {
  // At the very start of a function a new function scope will be created by JS engine
  // like functionScope = new FunctionScope();
  const c = {};
  function innerFunction() {
  }
  // functionScope.a -> string 'a'
  // functionScope.b -> number 10
  // functionScope.c -> object {}
  // functionScope.innerFunction -> function
}
// scope will be created only when function will be called
writeToScope('a', 10);
```

Global Context

Our code is executed inside a global function

For the global function its "scope" object it the same as "this" and is the "window".

```
// Somewhere in browser JS engine the function Global was called
// Global()
// It's scope object is a window object. But it also exposes "window" variable {
//   window: ...
//}
console.log(window === this); // -> true;
console.log(window.window.window === this); // -> true;
var x = 10;
console.log(window.x === x); // -> true;
console.log(window.x === this.x); // -> true;
```

Hoisting

Hoisting is a process of how variables appear in a function scope

1. "var", "let", "const" declarations go first

2. Function params go second and will overwrite params.

```
function tryHoisting(a) {
 // First: JS finds "var a" and hoists it declaration but it's value is undefined now.
 // Second: Function argument 'a' is written to functionScope.a -> string 'someText' - it's a param

 console.log(a); // -> writes "someText" to console

 var a = 10; // a is set to 10 instead of "someText";
 // functionScope.a -> is number 10 now
 console.log(a); // -> writes 10 to console
}
tryHoisting('someText');
```

Hoisting

3. Functions declarations go third and will overwrite params and vars (but will conflict with let or const);

4. But setting value to var will overwrite a function object

```
// "var hoisted" was raised first, but then was overwritten by function named "hoisted"
// typeof functionScope.hoisted -> "function"
hoisted(); // -> a function was called
console.log(hoisted); // -> Function body
var hoisted = 10;
// Now functionScope.hoisted = 10; typeof functionScope.hoisted -> "number"
console.log(hoisted); // -> 10
function hoisted() {
 console.log("a function was called");
}
```

Hoisting Again

1. var, let, const and function are hoisted
2. This is done to "hide" parent scope variables to avoid confusion (or to create one)
3. var is hoisted in a whole function scope

```
var parentPropertyVar = 'var';

function hoistingExample() {
  console.log('value: ', parentPropertyVar);  // undefined
  var parentPropertyVar = 'hidden var';
  console.log('value: ', parentPropertyVar); // 'hidden var'
}

console.log('value: ', parentPropertyVar); // var
hoistingExample();
```

Hoisting of Let and Const

1. let and const are hoisted in a block code scope:
   { ... }
2. Trying to access a variable before the declaration produces an error
3. Error occurs because variable is in so called Temporary Dead Zone

```
let parentPropertyLet = 'let';

function hoistingExample(hideParentProperty) {
   if (hideParentProperty) {
     // console.log('value: ', parentPropertyLet);  -> will produce an error
     let parentPropertyLet = 'hidden let';
     console.log('value: ', parentPropertyLet);
   } else {
     console.log('value: ', parentPropertyLet);
   }
}
hoistingExample(false);
hoistingExample(true);
```

Declaration, Expression, Self-invocation

Declared function is just like a simple variable in the current scope (can be called anywhere in the current scope)

```
printParam('print me!');
function printParam(param) {
 console.log(param)
}
```

Functional expression is a real variable (can be called only after variable gets the function object as a value)

```
let printParam = function (param) {
 console.log(param)
}
printParam('print me!');
```

Self-invoked function is called only at the place it was decalred

```
(function(param) {
 console.log(param)
})('print me!');
```

Function Declaration

has a required name;

in the source code position it is positioned: either at the Program level or directly in the body of another function (FunctionBody);

is created on entering the context stage;

influences variable object;

and is declared in the following way:

function exampleFunc() {

...

}

Local Variables

A variable declared inside a function is only visible inside that function.

```
function showMessage() {

  let message = "Hello, I'm JavaScript!"; // local variable

  alert( message );
}

showMessage(); // Hello, I'm JavaScript!

alert( message ); // <-- Error! The variable is local to the function
```

Outer Variables

A function can access an outer variable as well

The function has full access to the outer variable

```
let userName = 'John';

function showMessage() {

  userName = "Bob"; // (1) changed the outer variable

  let message = 'Hello, ' + userName;
  alert(message);
}

alert( userName ); // John before the function call

showMessage();

alert( userName ); // Bob, the value was modified by the function
```

Function Arguments

Values we are passed on a function invocation are called parameters or function arguments

When a function is called and new scope is created - it has predefined property "arguments"

Its a storage for the parameters which were passed on a function call;

```
function tryArguments(a, b) {
 // At the very start of function a new function scope will be created
 // functionScope = {
 //    arguments: pseudo-array of passed parameters
 // }
 // all variable declarations become its properties
 console.log(a); // the same like arguments[0]
 console.log(b); // the same like arguments[1]
 // It's possible to get passed value even if you have not declared a param name
 console.log(arguments[2]);

}
tryArguments(1, 2, 3);
```

Default Function Parameters

Default values (ES6)

Default function parameters allow formal parameters to be initialized with default values if no value or undefined is passed.

```
function showArg(arg = 'some default value') {
  console.log(arg);
}

showArg(1); // -> 1
showArg();  // -> some default value
```

Returning a Value

A function can return a value back into the calling code as the result.

Function returns value with return keyword.

You may use return anywhere in body, but it will stop execution of a function and jump out of it immediately.

```
function greet(name) {
 var greetMessage = 'Hello, ' + name + '!';
 return greetMessage;
}

console.log( greet('JS') ); // 'Hello, JS!'
```

```
function checkUser(login) {
 if (login == undefined) {
  return 'You did not specify a login';
 } else if (login == 'admin') {
  return 'It is admin';
 } else if (login == 'user') {
  return 'It is user';
 }
 return 'User is unknown';
}
```

Return

A function with an empty return or without it returns undefined

If a function does not return a value, it is the same as if it returns undefined

```
function doNothing() {
/* empty */
}
alert( doNothing() === undefined ); // true
```

```
function doNothing() {
  return;
}
alert( doNothing() === undefined );
```

Naming a function

Functions are actions. So their name is usually a verb.

Function starting with…

"get…" – return a value,

"calc…" – calculate something,

"create…" – create something,

"check…" – check something and return a boolean, etc.

showMessage(..) // shows a message

getAge(..) // returns the age (gets it somehow)

calcSum(..) // calculates a sum and returns the result

createForm(..) // creates a form (and usually returns it)

checkPermission(..) // checks a permission, returns true/false

Summary

Values passed to a function as parameters are copied to its local variables.

A function may access outer variables. But it works only from inside out. The code outside of the function doesn't see its local variables.

A function can return a value. If it doesn't then its result is undefined.

Function naming:

A name should clearly describe what the function does. When we see a function call in the code, a good name instantly gives us an understanding what it does and returns.

A function is an action, so function names are usually verbal.

There exist many well-known function prefixes like create…, show…, get…, check… and so on. Use them to hint what a function does.

Function Expression

in the source code can only be defined at the expression position;

can have an optional name;

it’s definition has no effect on variable object;

and is created at the code execution stage.

```
var foo = function () {
  ...
};
```

Function Expression

```
var foo = function () {
  ...
};
var foo = function _foo() {
  ...
};
(function () {
   // initializing scope
  })();
(function () {}());
```

```
// in parentheses (grouping operator) can be only an expression
(function foo() {});
 // in the array initialiser – also only expressions
[function bar() {}];
 // comma also operates with expressions
1, function baz() {};
```

Named Function Expression

Named Function Expression or, shortly, NFE, is a term for Function Expressions that have a name.

It allows to reference the function from inside itself.

It is not visible outside of the function.

```
let sayHi = function func(who) {
  if (who) {
    alert(`Hello, ${who}`);
  } else {
    func("Guest"); // use func to re-call itself
  }
};

sayHi(); // Hello, Guest

// But this won't work:
func(); // Error, func is not defined (not visible outside of the function)
```

Callback Functions

A callback is a function that is to be executed after another function has finished executing

functions can take functions as arguments, and can be returned by other functions. Functions that do this are called higher-order functions. Any function that is passed as an argument is called a callback function.

```
function greeting(name) {
  alert('Hello ' + name);
}

function processUserInput(callback) {
  var name = prompt('Please enter your name.');
  callback(name);
}

processUserInput(greeting);
```

Function Expression vs Function Declaration

FUNCTION DECLARATION

a function, declared as a separate statement, in the main code flow.

is usable in the whole script/code block.

FUNCTION EXPRESSION

a function, created inside an expression or inside another syntax construct.

is created when the execution reaches it and is usable from then on.

Arguments Object

```
function myConcat(separator) {
var result = "";
for (var i = 1; i < arguments.length; i++) {
result += arguments[i] + separator;
}
return result;
}
myConcat(", ", "red", "orange", "blue");
myConcat("; ", "elephant", "giraffe", "lion", "cheetah");
```

Default Parameters

```
function multiply(a, b) {
  b = typeof b !== 'undefined' ? b : 1;
  return a * b;
}
multiply(5);

function multiply(a, b = 1) {
  return a * b;
}
multiply(5);

function multiply(a, b = 2 * a) {
  return a * b;
}
multiply(5);
```

Rest Parameter

The rest parameters can be mentioned in a function definition with three dots ....

const myConcat = (sep, ...strings) => strings.join(sep)

Rest & Spred

REST

Rest parameters has to be the last one and you can't have more than one.

```
function func(...rest) {
  console.log(rest);
}
func('a', 'b', 1, 2);
```

SPRED

Spred operator can also be used with other paramters, multiple times as well as mixed.

```
var array = [2, 3, 5];
Math.min(...array);
```

```
function func() {
  console.log(arguments);
}
const parameters = [3, 4, 8];
func(1, ...parameters, 11, ...[15, 20]);
```

new Function

The syntax for creating a function:

let func = new Function ([arg1[, arg2[, ...argN]],] functionBody)

```
const sum = new Function('a', 'b', 'return a + b');

console.log(sum(2, 6)); //8
```

Closure

Usually, a function remembers where it was born in the special property [[Environment]]. It references the Lexical Environment from where it's created.

But when a function is created using new Function, its [[Environment]] references not the current Lexical Environment, but instead the global one.

```
function getFunc() {
  let value = "test";
  let func = new Function('alert(value)');
  return func;
}
getFunc()(); // error: value is not defined
```

```
function getFunc() {
  let value = "test";
  let func = function() {
    alert(value);
  };
  return func;
}
getFunc()(); // "test"
```

Summary

Functions are values. They can be assigned, copied or declared in any place of the code.

If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".

If the function is created as a part of an expression, it's called a "Function Expression".

Function Declarations are processed before the code block is executed. They are visible everywhere in the block.

Function Expressions are created when the execution flow reaches them.

Scope Again

Scope in an object, it's created at function call (activation object)

All variables and functions become properties of scope object

```
function parentFunction() {
  // parentFunction scope created. vairable and child function become its properties
  /*
    scope: {
     variable: undefined,
     childFunction: childFunction
    }
  */
  var variable = 10;
  function childFunction() {
     // Something useful here
  }

}
parentFunction();
```

Function Parent Scope

Function, as an object, has a special property [[Scope]].
When function is declared - two things happen:

1. function becomes a property of scope object

2. current scope object is written to [[Scope]] property of function object

```
function parentFunction() {
  // parentFunction scope created.
  /*
    1. Child function becomes its property
    scope: {
     childFunction: childFunction
    }

    2. Scope becomes a property of child function
    childFunction.[[Scope]] = scope
  */
  function childFunction() {
     // Something useful here
  }
}
parentFunction();
```

Closure

When function "knows" the scope where it was declared - we call it "closure"
As [[Scope]] is a property of function object itself - even if we return function and call it at any other place in the code - it will remember scope where it was declared

```
function parent() {
  let childCallCount = 0;
  function child() {
    childCallCount++;
    console.log(`called ${childCallCount} time(s)`);
  }

  child();
  return child;
}

let childFirst = parent(); // called 1 time(s)
childFirst(); // called 2 time(s)
childFirst(); // called 3 time(s)

let childSecond = parent();// called 1 time(s)
childSecond(); // called 2 time(s)
childSecond(); // called 3 time(s)
```

Scope Chain. Variable Lookup

Folded functions create a chain of scopes

When we are trying to access a variable - JS will seek it in current scope and up through the chain

We can "hide" the parent variable by declaration a new variable with the same name

If no variable declaration were found - exception will occur

```
function parent() {
  let parentProperty = 10;
  function child() {
    console.log(parentProperty);  // 10
  }
  child();

  function childHidesParentProperty() {
    let parentProperty = 20;
    function childOfAChild() {
      console.log(parentProperty); // 20
    }
    childOfAChild();
  }
  childHidesParentProperty();
}
```

Parent Scope is Shared Between Child Functions

It's really obvious, but let's see:

```
function getCounter() {
  let counter = 0;
  function increment(incrementBy) {
    if (typeof incrementBy === 'undefined') {
      incrementBy = 1;
    }
    counter = counter + incrementBy;
  }
  function decrement(decrementBy) {
    if (typeof decrementBy !== 'number') {
      decrementBy = 1;
    }
    counter = counter - decrementBy;
  }
  function printCounter() {
    console.log('Current value is: ' + counter);
  }
  return {
    inc: increment,
    dec: decrement,
    print: printCounter,
  };
}
```

```
let counter = getCounter();
counter.inc(10);
counter.print(); // 10
counter.dec();
counter.print(); // 9
```

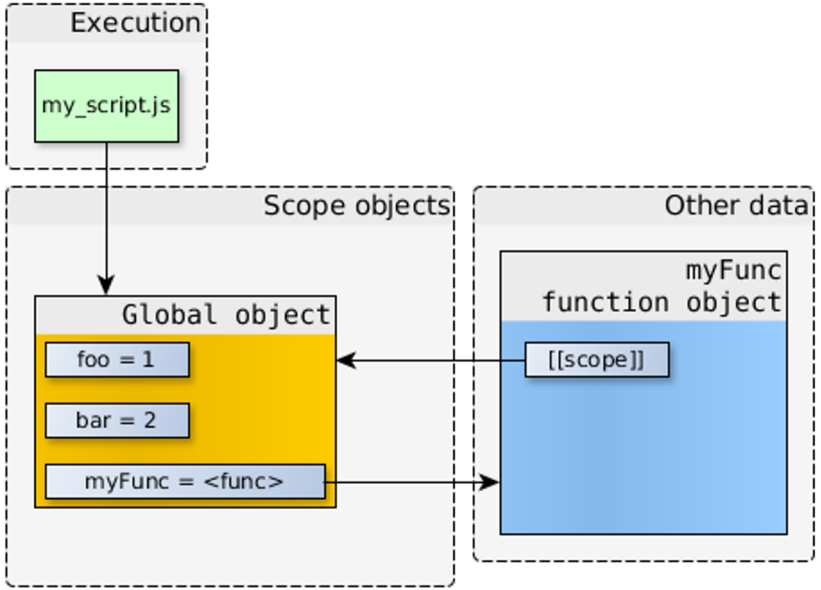
Parent Scope is Shared Between Child Functions ES6 Example

```
function getCounter() {
  let counter = 0;
  return {
    inc: (incrementBy = 1) => (counter += incrementBy),
    dec: (decrementBy = 1) => (counter -= decrementBy),
    print: () => console.log(`Current value is: ${counter}`),
  };
}
let counter = getCounter();
counter.inc(10);
counter.inc();
counter.print(); // 10
counter.dec();
counter.print(); // 9
```

About Scope Yet

A function records the scope it was created in via the internal property [[Scope]].

```
var foo = 1;
var bar = 2;
function myFunc() {
  //-- define local-to-function variables
  var a = 1;
  var b = 2;
  var foo = 3;
  console.log('inside myFunc');
}
console.log('outside');
//-- and then, call it:
myFunc();
```



Summarize

1. Each function object gets a link to a scope where it was declared - closure
2. JS seeks variable value up through scopes chain
3. If we declare variable with the same name in child scope it will hide parent variable from us. But only in current function.
4. Scope is shared between child functions, they will have access to parent variable

Call Stack

1. Call stack holds the ordered list of called functions.
2. When function returns the result - it is removed from stack
3. When the error occurs we can see the current call stack - useful for debugging

```
secondInStack();

function secondInStack() {
  thirdInStack();
}

function thirdInStack() {
  console.log("I'm the third!");
}
```

Recursion

Recursion is a programming pattern that is useful in situations when a task can be naturally split into several tasks of the same kind, but simpler.

When a function solves a task, in the process it can call many other functions. A partial case of this is when a function calls itself. That's called recursion.

Recursion algorithm

1. Recursion is an operation when function can call itself once again.
2. Can be useful for parsing, tree walking
3. Call stack is not endless
4. Any recursion can be changed to loop

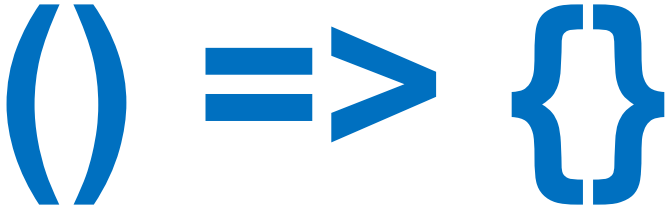
```
const usefulStorage = [1, 2, 3];

popFromStorage(usefulStorage);
// Item: 3
// Item: 2
// Item: 1
// Storage is empty!

function popFromStorage(storage) {
  let item = storage.pop();
  console.log("Item: ", item);
  if (storage.length > 0) {
    popFromStorage(storage);
  } else {
    console.log("Storage is empty!");
  }
}
```

Recursion Example

```
function pow(x, n) {
  if (n == 1) {
    return x;
  } else {
  return x *= pow(x,n - 1);
}}
alert( pow(2, 3) ); // 8
```

Arrow

An arrow function expression has a shorter syntax than a function expression and does not have its own this, arguments, super, or new.target. These function expressions are best suited for non-method functions, and they cannot be used as constructors.




```
(param1, param2, …, paramN) => { statements }
(param1, param2, …, paramN) => expression
// equivalent to: (param1, param2, …, paramN) => { return expression; }

// Parentheses are optional when there's only one parameter:
(singleParam) => { statements }
singleParam => { statements }

// A function with no parameters requires parentheses:
() => { statements }
() => expression // equivalent to: () => { return expression; }

let myVeryUsefulFunction = () => {
  console.log('meow');
};
```

It's always anonymous

Arrow Functions

There's one more very simple and concise syntax for creating functions, that's often better than Function Expressions. It's called "arrow functions"

let func = (arg1, arg2, ...argN) => expression

…This creates a function func that has arguments arg1..argN, evaluates the expression on the right side with their use and returns its result.

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

[0, 1, 2, 5, 10].map((x) => x * x * x); //[0, 1, 8, 125, 1000]

Arrow functions do not have "arguments"

Arrow Functions

Arrow functions are handy for one-liners. They come in two flavors:

Without figure brackets: (...args) => expression – the right side is an expression: the function evaluates it and returns the result.

With figure brackets: (...args) => { body } – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

Example

```
var an = (a, b) => a + b;
// or
var an = (a, b) => { return a + b };
// if we only have one parameter we can loose the parentheses
var an = a => a;
// and without parameters
// this looks pretty nice when you change something like:
[1,2,3,4].filter(function (value) {return value % 2 === 0});
// to:
[1,2,3,4].filter(value => value % 2 === 0);
```

Basic Syntax with One Parameter

Parentheses are optional when only one parameter is present

```
//ES5
var phraseSplitterEs5 = function phraseSplitter(phrase) {
  return phrase.split(' ');
};

//ES6
var phraseSplitterEs6 = phrase => phrase.split(' ');

console.log(phraseSplitterEs6('ES6 Awesomeness')); // ["ES6", "Awesomeness"]
```

No Parameters

Parentheses are required when no parameters are present

```
//ES5
var docLogEs5 = function docLog() {
  console.log(document);
};

//ES6
var docLogEs6 = () => {
  console.log(document);
};
docLogEs6(); // #document... <html> ….
```

No Separate this

An arrow function does not have its own this; the this value of the enclosing execution context is used

```
function Person() {
  this.age = 10;
  setInterval(() => {
    this.age++; // |this| properly refers to the person object
  }, 1000);
}

var p = new Person();
```

Array Methods

filter()
sort()
map()
reduce()

filter()

The filter() method creates a new array with all elements that pass the test implemented by the provided function.

filter() calls a provided callback function once for each element in an array, and constructs a new array of all the values for which callback returns a value that coerces to true. callback is invoked only for indexes of the array which have assigned values

callback is invoked with three arguments:

1. the value of the element
2. the index of the element
3. the Array object being traversed

```
function isBigEnough(value) {
 return value >= 10;
}
var filtered = [12, 5, 8, 130, 44].filter(isBigEnough);
```

sort()

The sort() method sorts the elements of an array in place and returns the array. The sort is not necessarily stable. The default sort order is according to string Unicode code points.

```
let arr = [ 1, 2, 15 ];

// the method reorders the content of arr (and returns it)

arr.sort();
console.log(arr); // [1, 15, 2]
```

Own Sorting Order

```
function compareNumeric(a, b) {
  if (a > b)  return 1;
  if (a === b) return 0;
  if (a < b) return -1;
}

let arr = [ 1, 22, 15 ];
arr.sort(compareNumeric);
console.log(arr);
```

```
var numbers = [4, 2, 5, 1, 3];
numbers.sort(function (a, b) {
  return a - b;
});
console.log(numbers);
```

map()

The map() method creates a new array with the results of calling a provided function on every element in the calling array

Return value

A new array with each element being the result of the callback function.

```
let result = arr.map(function (item, index, array) {
  // returns the new value instead of item
});

let lengths = ['Bilbo', 'Gandalf', 'Nazgul'].map(item => item.length);
console.log(lengths); // 5,7,6
```

reduce()

The reduce() method applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value.

reduce executes the callback function once for each element present in the array, excluding holes in the array, receiving four arguments:

- accumulator
- currentValue
- currentIndex
- array

How reduce Works

```
[0, 1, 2, 3, 4].reduce(
  function (  accumulator,  currentValue,  currentIndex,  array  ) {
    return accumulator + currentValue;
  }
);
```

callback	accumulator	currentValue	currentIndex	array	return value
first call	0	1	1	[0, 1, 2, 3, 4]	1
second call	1	2	2	[0, 1, 2, 3, 4]	3
third call	3	3	3	[0, 1, 2, 3, 4]	6
fourth call	6	4	4	[0, 1, 2, 3, 4]	10

Examples

SUM ALL THE VALUES OF AN ARRAY

```
var sum = [0, 1, 2, 3].reduce(function (a, b) {
  return a + b;
}, 0); // sum is 6
```

ALTERNATIVELY, WRITTEN WITH AN ARROW FUNCTION:

```
var total = [0, 1, 2, 3].reduce((acc, cur) => acc + cur, 0);
```

Scheduling a Call

setTimeout allows to run a function once after the interval of time.

setInterval allows to run a function regularly with the interval between the runs.

setTimeout

```
function sayHi() {
  alert('Hello');
}

setTimeout(sayHi, 1000);
```

```
function sayHi(phrase, who) {
  alert( phrase + ', ' + who );
}

setTimeout(sayHi, 1000, "Hello", "John"); // Hello, John
```

Canceling with clearTimeout

A call to setTimeout returns a “timer identifier” timerId that we can use to cancel the execution.

```
let timerId = setTimeout(() => alert("never happens"), 1000);
alert(timerId); // timer identifier

clearTimeout(timerId);
alert(timerId); // same identifier (doesn't become null after canceling)
```

setInterval

All arguments have the same meaning. But unlike setTimeout it runs the function not only once, but regularly after the given interval of time.

To stop further calls, we should call clearInterval(timerId).

```
// repeat with the interval of 2 secondslet
timerId = setInterval(() => alert('tick'), 2000);

// after 5 seconds stop

setTimeout(() => { clearInterval(timerId);
alert('stop'); }, 5000);
```

Global Context. this

In the global execution context (outside of any function), this refers to the global object whether in strict mode or not.

```
console.log(this === window); // true
```

Function Execution Context. this

The environment in which function is executed is called function execution context
It consists of:

Function scope

Function parent scope (the whole chain)

"arguments" property

"this" keyword - often is called "function context"

Exact value of "this" is defined when the function is called

```
function showContext() {
  console.log(this); // 'window' in browser
}
showContext();
```

When Function is a Method of an Object

We say "object has a method" when a value of its property is a function

someObject.method() - is just a way to set a function context

```
const counter = {
  currentValue: 100,
  print: function() {
    console.log(this.currentValue);
  }
}

counter.print();  // function context is "counter" object

let printFunction = counter.print;
printFunction();  // function context is Window
```

When Function is a Method of an Object (Example)

```
'use strict';

var myObj = {
  myProp: 100,
  myFunc: function myFunc() {
    return this.myProp;
  },
};
console.log(myObj.myFunc());
```

```
function MyObj() {
  this.a = 'a';
  this.b = 'b';
}
var myObj = new MyObj();

console.log(myObj);
```

When Function is a Constructor

We say "constructor function" when we use operator "new" with it and receive a new object instance

Is is almost the same like to use .call({}, ...);

```
function Counter(initialValue) {
  this.currentValue = initialValue,
  this.print = function() {
    console.log(this.currentValue);
  }
}

const counter = new Counter(50);
counter.print(); // function context is "counter" object
```

ES6 Arrow Function and Execution Context

Arrow function is always "bound" to current value of this

It has no "arguments" object

```
function Counter(initialValue) {
  this.currentValue = initialValue,
  this.print = () => {
    console.log(this.currentValue);
  }
}

const counter = new Counter(50);
counter.print();  // function context is "counter" object
let printFunction = counter.print;
printFunction();  // function context is still a "counter" object
```

Summary

Functions that are stored in object properties are called “methods”.

Methods allow objects to “act” like object.doSomething().

Methods can reference the object as this.

The value of this is defined at run-time.

When a function is declared, it may use this, but that this has no value until the function is called.

That function can be copied between objects.

When a function is called in the “method” syntax: object.method(), the value of this during the call is object.

Promise

The Promise object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value.

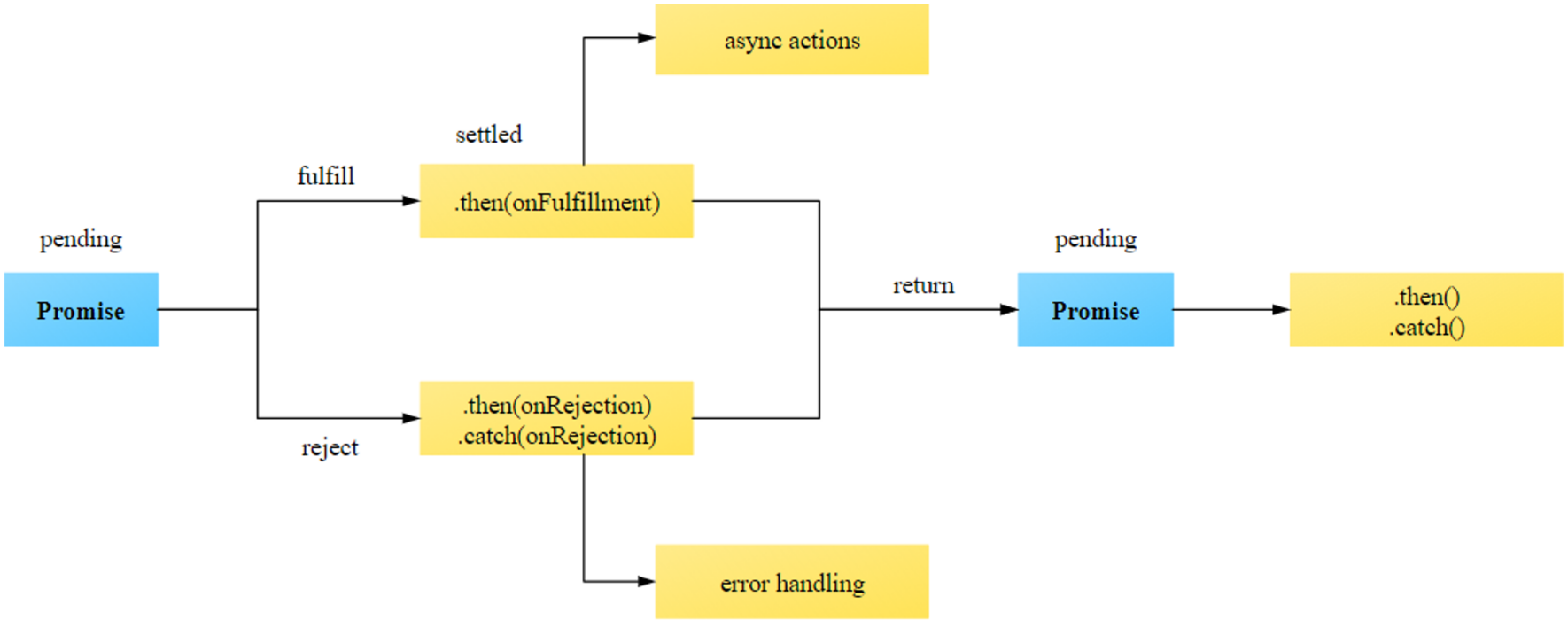
A Promise is in one of these states:

pending: initial state, neither fulfilled nor rejected.

fulfilled: meaning that the operation completed successfully.

rejected: meaning that the operation failed.

Promise


async actions
settled
fulfill
.then(onFulfillment)
pending
Promise
return
pending
Promise
.then()
.catch()
.then(onRejection)
.catch(onRejection)
reject
error handling

Async/await

The async function declaration defines an asynchronous function, which returns an AsyncFunction object.

When an async function is called, it returns a Promise. When the async function returns a value, the Promise will be resolved with the returned value. When the async function throws an exception or some value, the Promise will be rejected with the thrown value.

await

An async function can contain an await expression, that pauses the execution of the async function and waits for the passed Promise's resolution, and then resumes the async function's execution and returns the resolved value

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Ready!"), 1000)
  });

  let result = await promise; // will wait until the promise is fulfilled (*)

  alert(result); // "Ready!"
}

console.log(f());
```

Useful Links

https://developer.mozilla.org/ru/docs/Web/JavaScript/Closures
https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/this
https://developer.mozilla.org/ru/docs/Glossary/Call_stack
http://dmitrysoshnikov.com/ecmascript/chapter-6-closures/

Q&A

UA Frontend Online LAB