Wybrane elementy praktyki projektowania oprogramowania

Wykład 04/15
JavaScript, funkcje

Wiktor Zychla 2019/2020

1 Spis treści

2 Funkcje 2
2.1 Przekazywanie argumentów do funkcji 2
2.2 Zmienne lokalne 2
2.3 Zwracanie funkcji z funkcji i przekazywanie funkcji do funkcji, domknięcia 2
2.3.1 Przykład 1 2
2.3.2 Przykład 2 – memoizacja 3
2.3.3 Przykład 3 3
2.3.4 Przykład 4 – IIFE 5
2.4 this, call, apply, bind 5
3 Iteratory, generatory 8

2 Funkcje

Javascript jako język funkcyjny posługuje się pojęciem funkcji jako obiektem pierwszoklasowym.

Podczas wykładu omówimy następujące elementy programowania funkcyjnego:

2.1 Przekazywanie argumentów do funkcji

- Domyślne wartości argumentów
- Zmienna liczba argumentów
    - Za pomocą obiektu arguments
    - Za pomocą operatora rozrzucania – w tym miejscu zrobimy szerszą dygresję o przypisaniach destrukturyzyjących

2.2 Zmienne lokalne

- Hoisting
- Domyślny zasięg zmiennych – funkcyjny
- Zasięg blokowy – var vs let

2.3 Zwracanie funkcji z funkcji i przekazywanie funkcji do funkcji, domknięcia

Po podstawowym przykładzie programowania funkcyjnego, skupimy się na mechanizmie domknięć (closures)

Omówione zostaną podstawowe przykłady domknięć:

2.3.1 Przykład 1

Funkcja która może zostać wywołana dowolną liczbę razy – tu domknięcie jest użyte do zapamiętania akumulatora:

```
function sump(x) {

    var _sum = x;
    var _f = function(y) {
        _sum += y;
        return _f;
    }

    _f.valueOf = function() {
        return _sum;
    }

    return _f;
}

console.log( sump(4)(5)(6) + 1 );
```

2.3.2 Przykład 2 – memoizacja

```
function fac(n) {

    if ( n > 0 ) {
        return n*fac(n-1);
    } else {
        return 1;
    }
}

function memoize(fn) {

    var cache = {};

    return function(n) {

        if ( n in cache ) {
            return cache[n]
        } else {
            var result = fn(n);
            cache[n] = result;
            return result;
        }

    }
}

var memofac2 = memoize(fac);

// pierwsze wyliczenie – napełni cache
console.log( memofac2(6) );
// drugie wyliczenie – z cache
console.log( memofac2(6) );
```

Przy okazji tego przykładu można spróbować odpowiedzieć na pytania:

- Jak uogólnić funkcję memoizującą tak, żeby radziła sobie z memoizowaniem funkcji o dowolnej liczbie parametrów?
- W powyższym przykładzie, dodając na końcu wyliczanie dla argumentu 5, można zaobserwować pewne niekoniecznie pożądane zjawisko - wartość zostanie owszem wyliczona, umieszczona w cache, ale przecież wcześniej, przy wyliczaniu wartości dla 6 już liczono raz wartość dla 5. Dlaczego funkcja memoizująca nie zapamiętuje wyników pomocniczych obliczeń? Jak sobie z tym poradzić w przypadku konkretnej funkcji? A jak poradzić sobie w ogólnym przypadku?

2.3.3 Przykład 3

Częściowa aplikacja funkcji

```
function sumpartial( x ) {
    return function( y ) {
        return x + y;
    }
}

var sum1 = sumpartial(1);

console.log( sum1(2) );
```

Ogólniejsze podejście umożliwia utworzenie częściowej aplikacji funkcji z dowolnego wywołania, tzw. częściowa aplikacja funkcji

```
function sum2(x,y) {
    return x + y;
}

function partial(fn, ...args ) {
    return function( ...brgs ) {
        return fn( ...args, ...brgs );
    }
}

var sum1 = partial(sum2,1);

console.log( sum1(2) );
```

Najogólniejsze podejście to tzw. rozwijanie funkcji (currying) w którym częściowa aplikacja jest możliwa dla każdego argumentu z osobna (wywołanie z n argumentami zamienia się na n wywołań z jednym argumentem)

```
function sum3(x,y,z) {
    return x + y + z;
}

function rec(fn, i, args) {
    if ( fn.length == 0 ) return fn;

    if ( i < fn.length ) {
        return (x) => {
            args.push(x);
            return rec(fn, i+1, args);
        }
    } else {
       return fn(...args);
    }
```

```
}

function curry(fn) {
    return rec(fn, 0, []);
}

var currysum3 = curry(sum3);

console.log( currysum3(1)(2)(3) );
```

2.3.4 Przykład 4 – IIFE

Technika IIFE (Immediately-Invoked Function Expression) jest symptomatyczna dla funkcyjnego stylu programowania w Javascript. Umożliwia wykonanie części pracy funkcji i ukrycie w zasięgu funkcji szczegółów implementacyjnych (w tym np. pomocniczych struktur danych). Typowo tej techniki używa się np. do bloków inicjujących.

```
var counter = (function () {
    var i = 0;

    return {
        get: function () {
            return i;
        },
        increment: function () {
            return ++i;
        }
    };
})();

console.log(counter.get());
counter.increment();
counter.increment();
console.log(counter.get());
```

W powyższym przykładzie, zmienna lokalna **i** jest współdzielona między metodami **get** i **increment a** ponieważ trafia w ich domknięcie to nie jest widoczna na zewnątrz. Z kolei dzięki IIFE, obiekt **counter** jest od razu gotowy do użycia, ponieważ za jego zainicjowanie odpowiada wartość funkcji która jest natychmiast wywołana po zadeklarowaniu.

2.4 this, call, apply, bind

W przeciwieństwie do innych języków z jednoznacznym **this**, które oznacza obiekt – właściciel wywołanej metody, w Javascript **this** zależy od sposobu wywołania funkcji.

W funkcji wywołanej bez tzw. wiązania, **this** odnosi się do obiektu globalnego

```
function foo() {
    return this.x; // ?
}

console.log( foo() );
```

W funkcji wywołanej z wiązaniem na obiekcie, **this** ma wartość referencji do tego obiektu

```
var person = {
    name: 'jan',
    say: function() {
        return `${this.name}`
    }
}

console.log( person.say() );
```

O ile ten przykład może przywoływać intuicję z innych języków o tyle warto go uzupełnić rozszerzeniem, w którym ta sama metoda zostanie wywołana bez wiązania – wtedy obowiązuje pierwsza zasada:

```
var person = {
    name: 'jan',
    say: function() {
        return `${this.name}`
    }
}

var _f = person.say;

console.log( _f() ); // brak wiązania!
```

Jak w takim razie wymusić wiązanie **this** w funkcji do wybranego kontekstu? Otóż okazuje się, że funkcję można wywołać nie bezpośrednio, tylko za pomocą **apply/call**, w którym pierwszym argumentem jest … kontekst **this** w funkcji!

```
var person = {
    name: 'jan',
    say: function() {
        return `${this.name}`
    }
}

var _f = person.say;

console.log( _f() ); // brak wiązania!
console.log( _f.call(person) ); // jawne wiązanie!
```

Różnica między apply a call polega na tym że rzeczywiste argumenty wywołania są podawane albo przez tablicę (apply) albo przez listę argumentów oddzielonych przecinkiem (call)

```
function foo(y,z) {
    return this.x + y + z;
}

var o = { x : 1 }

// apply - jeden argument: tablica argumentów
console.log( foo.apply( o, [1, 2] ) );

// call - lista argumentów
console.log( foo.call( o, 1, 2 ) );
```

Z kolei bind to operator który z funkcji zwraca funkcję o tej samej sygnaturze, ale z dowiązanym **this:**

```
function foo(y,z) {
    return this.x + y + z;
}

var o = { x : 1 }

var _foo = foo.bind(o);

console.log(_foo(1,2));
```

3 Iteratory, generatory

Iterator to funkcja bezargumentowa która zwraca obiekt, który ma jedno pole, **next**, które jest funkcją zwracającą obiekt o polach **value** i **done.** W naiwnej implementacji

```
function createIterator() {
    var _state = 0;
    return {
        next : function() {
            return {
                value: _state,
                done: _state++ >= 10
            }
        }
    }
}

var it = createIterator();

var _result;
while ( _result = it.next(), !_result.done ) {
    console.log( _result.value );
}

it = createIterator();
for ( var _res; _res = it.next(), !_res.done; ) {
    console.log( _res.value );
}
```

Funkcja iteratora jeśli zostanie użyta w obiekcie jako wartość składowej **Symbol.iterator**, obiekt uzyskuje możliwość iterowania jego zawartości za pomocą **for-of**.

```
var foo = {
    [Symbol.iterator]: createIterator
}

for ( var f of foo ) {
    console.log(f);
}
```

Warto w tym miejscu przyjrzeć się co się dzieje jeśli sama funkcja generatora nie jest funkcją bezargumentową – jak wtedy zmienić jej użycie w [**Symbol.iterator**]?

Generator to skrócony sposób zapisu kodu iteratora, dodający zwracanie bieżącej wartości za pomocą **yield.** Iterowanie generatora działa tak samo jak iteratora

```
function* createGenerator() {
    for ( var i=0; i<10; i++ ) {
```

```
        yield i;
    }
}

it = createGenerator();

for ( var _res; _res = it.next(), !_res.done; ) {
    console.log( _res.value );
}

var bar = {
    [Symbol.iterator]: createGenerator
}

for ( var b of bar ) {
    console.log( b );
}
```