

Wybrane elementy praktyki projektowania oprogramowania

Wykład 03/15

JavaScript, funkcje

Wiktor Zychla 2023/2024

1 Spis treści

2	Funkcje	2
2.1	Przekazywanie argumentów do funkcji.....	2
2.2	Zmienne lokalne	4
2.3	Zwracanie funkcji z funkcji i przekazywanie funkcji do funkcji, domknięcia	4
2.3.1	Wprowadzenie	4
2.3.2	Przykład 1 – funkcja „nieskończona”	5
2.3.3	Przykład 2 - funkcja samomodyfikująca się	5
2.3.4	Przykład 3 – memoizacja	6
2.3.5	Przykład 4 – częściowa aplikacja, currying	7
2.3.6	Przykład 5 – IIFE.....	8
2.4	this, call, apply, bind	10
3	Iteratory, generatory	13
3.1	Iterator	13
3.2	Generator	13
3.3	Przykład	14

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

- Funkcję zdefiniowaną z określoną liczbą parametrów można wywołać z dowolną liczbą argumentów
- Składnia definicji funkcji

```
function foo(n) {  
    return n + 1;  
}  
  
var bar = function(n) {  
    return n + 1;  
}  
  
var qux = n => n + 1;  
  
console.log( foo(1), bar(1), qux(1) );
```

- Domyślne wartości argumentów

```
function defaultArgs(a=1, b=2) {  
    console.log(a,b);  
}  
  
defaultArgs();
```

- Wywoływanie z pominiętymi argumentami

```
function omittedArgs(a, b, c, d, e) {  
    console.log(a,b,c,d,e);  
}  
  
omittedArgs(...[,],1,...[,]);
```

- Zmienna liczba argumentów

- Za pomocą obiektu [arguments](#) to najogólniejszy sposób uzyskiwania tzw. przeciążania funkcji – funkcja jest jedna ale reaguje na różne sposoby jej wywołania

```
/* funkcja reaguje na wywołania
   overload(number)
   overload(string, string)
*/
function overload(a,b) {
    if ( arguments.length == 1 && typeof a == 'number' ) {
        console.log( `overload(int: ${a})` );
    }
    else
    if ( arguments.length == 2 &&
        typeof a == 'string' && typeof b == 'string' )
    {
        console.log( `overload(string: ${a}, string: ${b})` );
    }
    else
    {
        console.log( 'nierozpoznane wywołanie' );
    }
};

overload(1);
overload('a', 'b');
overload(1, 'a');
```

- [Za pomocą operatora rozrzucania](#) – w tym miejscu zrobimy szerszą dygresję o [przypisaniach destrukuryzujących](#)

```
function f({name, surname, address: {city, number}={}}) {
    console.log(name);
    console.log(surname);
    console.log(city);
    console.log(number);
}

f({});
f({name:"jan"});
f({name:"jan", surname: "kowalski", address:{city:'wroclaw', number: 17}})
```

```
function returnTwoValues() {
    return [1,2];
}
var [a,b] = returnTwoValues();

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

2.2 Zmienne lokalne

- [Hoisting](#)
- Domyślny zasięg zmiennych – funkcyjny, w przeciwieństwie do blokowego, do którego jesteśmy przyzwyczajeni z innych języków
- Zasięg blokowy – [var vs let](#)

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

2.3.1 Wprowadzenie

Funkcje przekazywane do funkcji jako parametr, funkcje zwracane z funkcji jako wartość:

```
// funkcja przekazana jako argument do innej
function apply(f, n) {
    return f(n);
}

console.log( apply( x => x + 1, 1 ) );

// funkcja zwrócona jako wartość z innej
function create() {
    return function() {
        return 1;
    }
}

console.log( create()() ); // dlaczego ()() ?
```

Ten styl programowania jest nieodłączną cechą biblioteki standardowej

```
var a = [1,2,3,4];

console.log( a.find( x => x >= 2 ) );
console.log( a.map( x => x + 1 ) );
console.log( a.sort( (x,y) => y-x ) );
```

Częściowa aplikacja funkcji – tu po raz pierwszy można zaobserwować **domknięcie** ([closure](#)).

```
function sumpartial( x ) {  
    return function( y ) {  
        return x + y;  
    }  
}  
  
var sum1 = sumpartial(1);  
  
console.log( sum1(2) );
```

2.3.2 Przykład 1 – funkcja „nieskończona”

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) );
```

Proszę zwrócić uwagę na wykorzystane unarnego operatora + do konwersji czegoś (w tym przypadku *funkcji*) na wartość liczbową.

2.3.3 Przykład 2 - funkcja samomodyfikująca się

Jedna ze stosowanych technik programistycznych w Javascript jest nadpisanie referencji do funkcji przez inną definicję, w trakcie działania aplikacji. W szczególnych przypadkach funkcja może sama się zmodyfikować:

```
function f(s) {
```

```

    if ( s == "bar" ) {
        f = function(s) {
            return s + " zmienione";
        }
    }

    return s + " przed zmianą";
}

console.log( f('foo') );
console.log( f('bar') );
console.log( f('qux') );

```

2.3.4 Przykład 3 – 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?

2.3.5 Przykład 4 – częściowa aplikacja, currying

Wróćmy do przykładu z wprowadzenia. 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) );

```

2.3.6 Przykład 5 – 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.

Wróćmy do przykładu rozwijania funkcji. Jedną z wad zaprezentowanego wyżej podejścia było to że funkcja pomocnicza **rec** jest globalna, podobnie jak **curry**.

W pierwszym podejściu po prostu schowamy **rec** wewnątrz **curry**:

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

function curry(fn) {
    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);
        }
    }

    return rec(fn, 0, []);
}

var currysum3 = curry(sum3);

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

To krok w dobrą stronę. Funkcja **rec** nie jest już dostępna publicznie. To podejście ma jednak nadal pewną wadę – każde wywołanie **curry** tworzy kopię **rec** która zajmuje pamięć.

Jedną ze stosowanych technik programistycznych w Javascript jest więc taka definicja funkcji, która wykorzystuje IIFE do zdefiniowania tejże funkcji:

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

const curry = (function() {
    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, []);
    }

    return curry;
})();

var currysum3 = curry(sum3);

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

```

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 (**o.funkcja()**), **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

3.1 Iterator

[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);
}
```

3.2 Generator

[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 );
}
```

3.3 Przykład

Poniżej najbardziej rozbudowany przykład do przestudiowania we własnym zakresie. Przykład łączy kilka omawianych do tej pory technik:

- Za tworzenie obiektu odpowiedzialna jest osobna funkcja. To dobry sposób na zamknięcie w jednym miejscu „przepisu” na tworzenie „podobnych” obiektów. W językach programowania opartych na klasach byłby to konstruktor i o tym jaki jest w Javascript odpowiednik konstruktora porozmawiamy na kolejnym wykładzie – wtedy będzie można przepisać ten kod poniżej na docelowo, jeszcze bardziej elegancko
- Do obiektu, do właściwości **[Symbol.iterator]** przypięta jest funkcja generująca iterator. Ta funkcja wymaga argumentu (wskazania węzła) ale sam iterator nie może mieć argumentu. Tę niezgodność rozwiązuje się łatwo przez domknięcie – funkcja **nodeIterator** ma argument ale zwraca funkcję bezargumentową

```
/* funkcja tworzy węzeł drzewa binarnego
   z podanej wartości (val), lewego i prawego poddrzewa
   węzłowi przypisuje funkcję iterującą określoną przez generator
*/
function createNode(val, left, right) {
    var node = {
        val,
        left,
        right };
    node[Symbol.iterator] = nodeIterator(node)
    return node;
}
```

```

/* generator iteratora dla przechodzenia drzewa binarnego wgłąb
funkcja generatora jeśli ma być użyta do enumeracji for-of
(Symbol.iterator)
nie może mieć argumentów
skorzystamy więc z domknięcia żeby przekazać jej argument przez funkcję
w której ją zanurzymy
*/
function nodeIterator(node) {
    return function*() {
        yield node.val;
        if ( node.left ) yield* node.left;
        if ( node.right ) yield* node.right;
    }
}

// twórz drzewo
//      1
//     / \
//    2   5
//   / \
//  3   4
var root =
    createNode(
        1,
        createNode(
            2,
            createNode(
                3),
            createNode(
                4)
        ),
        createNode(
            5) );

for ( var e of root ) {
    console.log(e);
}

```