Wybrane elementy praktyki projektowania oprogramowania

Wykład 06/15
JavaScript: modularność, programowanie asynchroniczne

Wiktor Zychla 2021/2022

1 Spis treści

2 Struktura kodu

Na poprzednich wykładach przedstawiono komplet informacji niezbędnych do symulowania znanych z innych języków programowania elementów struktury kodu:

2.1 Klasy

Inne języki obiektowe: klasy (`class`)

JavaScript: funkcje konstruktorowe lub lukier syntaktyczny (`class`)

2.2 Składowe prywatne

Inne języki obiektowe: `private`

JavaScript: stosowanie domknięć. Naiwnie, wprost za pomocą domknięcia, z ceną – dodatkowym zużyciem pamięci z powodu konieczności przechowania funkcji mającej mieć dostęp do składowej prywatnej w każdym nowym obiekcie:

```
function Person(name) {
    var _name;

    this.getName = function() {
        return _name;
    }

    _name = name;
}

var p1 = new Person('jan');
var p2 = new Person('tomasz');

console.log( p1.getName() );
console.log( p2.getName() );

console.log( p1._name ); // brak dostępu
```

Lepiej – wykorzystując fakt że funkcja **Symbol** tworzy unikalny obiekt nawet wtedy kiedy jest wołana drugi raz z tym samym argumentem:

```
var Person = (function() {

    // only Person can access nameSymbol
    var nameSymbol = Symbol('name');

    function Person(name) {
        this[nameSymbol] = name;
```

```
    }

    Person.prototype.getName = function() {
        return this[nameSymbol];
    };

    return Person;
}());

var p1 = new Person('jan');
var p2 = new Person('tomasz');

console.log( p1.getName() );
console.log( p2.getName() );

// nie ma sposobu żeby spoza Person dostać się do nameSymbol
// więc nie można z obiektu wydobyć ustawionej wartości
```

Przy okazji powinno się wyjaśnić dlaczego mamy **Symbol.iterator** a nie **Symbol('iterator')**

2.3 Przestrzenie nazw (*namespaces*)

Inne języki: `namespace`

JavaScript: zagnieżdżone obiekty

```
UWr = {}
UWr.weppo = {};
UWr.weppo.Person = function(name) {
    this.name = name;
}

var p = new UWr.weppo.Person('jan');

console.log( p.name );
```

Aby unikać redeklaracji obiektu (a więc: sprawdzania za każdym razem czy przypadkiem już istnieje!), można zastosować jakiś wzorzec struktury kodu, np. wykorzystując IIFE:

```

(function(uwr) {
    uwr.Person = function(name) {
        this.name = name;
    }
})( global.UWr = global.UWr || {} );

(function(uwr) {
```

```
    uwr.Worker = function(name) {
        this.name = name;
    }
})( global.UWr = global.UWr || {} );

var p = new UWr.Person('jan');
var w = new UWr.Worker('Tomasz');

console.log( p.name );
console.log( w.name );
```

2.4 Moduły / pakiety / zestawy

Inne języki: pakiety (JAR), zestawy, biblioteki współdzielone (*.dll)

JavaScript: moduły

JavaScript inaczej obsługuje moduły w przeglądarce (strona HTML asynchronicznie podczytuje kolejne pliki *.js załączone przez `<script src… />` i tu przydaje się wzorzec przestrzeni nazw omówiony wcześniej), a inaczej w środowisku node.js gdzie zaimplementowano synchroniczne moduły, załączane za pomocą `require`. Warto mieć świadomość jak w praktyce jest to zaimplementowane.

Warto zauważyć, że ustawienie wartości zwrotnej we właściwym momencie pozwala nawet na osiągnięcie efektu cykli w zależnościach między modułami:

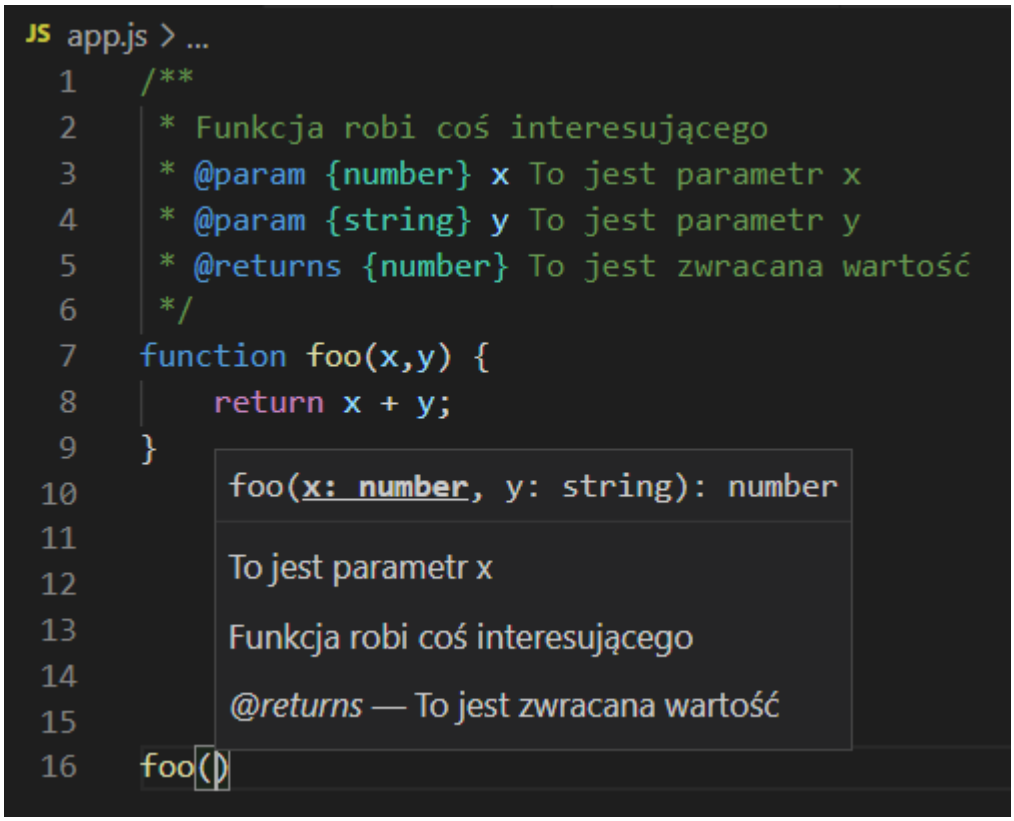
```
// main.js
let a = require('./a');

a.work_a(5);


// a.js
module.exports = { work_a };

let b = require('./b');

function work_a(n) {
    if ( n > 0 ) {
        console.log( `a: ${n}` );
        b.work_b(n-1);
    }
}

// b.js
module.exports = { work_b };

let a = require('./a');
```

```
function work_b(n) {
    if ( n > 0 ) {
        console.log( `b: ${n}` );
        a.work_a(n-1);
    }
}
```

W powyższym kodzie jest to osiągnięte przez ustawienie referencji do zwracanej wartości na początku modułu, podobny efekt dawałoby również ustawianie tej wartości lokalnie, wewnątrz funkcji która wymaga zależności.

2.5 Dokumentacja

Javascript ma dobrze ugruntowany standard dokumentacji kodu, JSDoc, którego stosowanie rekomenduje się o tyle, że edytory kodu potrafią korzystać z załączonych komentarzy, również wtedy kiedy funkcje są importowane z modułów:



3 Programowanie asynchroniczne

3.1 Historia paradygmatu

- dlaczego programowanie synchroniczne jest tak naprawdę iluzją? Bo io/sieć są z natury asynchroniczne
- w rzeczywistości synchroniczne interfejsy programowania (np. fopen/fread) powodują niepotrzebne znaczne obniżenie wydajnosci kodu (bo w czasie kiedy czeka na wyniki, procesor mógłby robić wiele innych rzeczy) – w czasach kiedy środowisko użytkownika było jednowątkowe a procesor miał jeden rdzeń to nie miało większego znaczenia. Ale w środowisku wielowątkowym, zwłaszcza takim gdzie maszyna jest **serwerem** wielu równoczesnych żądań, blokowanie rdzenia procesora na oczekiwanie na dane jest poważnym marnotrawstwem (!)
- kod asynchroniczny powoduje lepsze wykorzystanie zasobów maszyny (nie czeka się bez potrzeby) …
- … ale jest to trudne syntaktycznie – np. C/C++ nie odważyły się na propozycję składni asynchronicznej
- przełomem tym na który czekano całe lata jest pomysł "compiling with continuations":
    - w C#/.NET: **Task**,
    - w Javascipt: **Promise**,
    - w python: **asyncio.coroutine**

  który pozwala na pisanie kodu asynchronicznego który syntaktycznie jest możliwe najbliżej synchronicznego (przykład)

3.2 Pętla zdarzeń

Środowisko uruchomieniowe Javascript jest domyślnie jednowątkowe. Poszczególne funkcje są wykonywane synchronicznie. Operacje asynchroniczne są kolejkowane w ramach w ramach tzw. pętli zdarzeń (*event loop*).

Bardzo czytelne wyjaśnienie - prezentacja z konferencji JSConf'14. Pętla zdarzeń działa tak samo w każdym środowisku JS, w przykładzie pokazana jest przeglądarka, środowisko uruchomieniowe node.js oparte jest na tej samej zasadzie.

Event loop można sobie nieformalnie wyobrazić jako listę funkcji (callbacków) oczekujących na wykonanie

```
// pseudokod pętli zdarzeń (event loop)
var tasks = [];
while (tasks.length) {

    1) pobierz kolejną funkcję do wykonania z listy
    2) wykonaj funkcję i usuń z listy
    3a) jeśli funkcja wykonała kod który potencjalnie może powodować
        pojawienie się kolejnych funkcji na liście - czekaj
    3b) (node.js) jeśli funkcja nie wykonała kodu który potencjalnie może
        powodować pojawienie się kolejnych funkcji na liście - zakończ
    4) wznów przetwarzanie po pojawieniu się kolejnej funkcji
}
```

Najprostszym sposobem skierowania kodu do pętli zdarzeń jest użycie **setImmediate**/**setTimeout**:

```
setImmediate( () => {
    console.log("a");
});
console.log("b");
```

W przeglądarce można użyć funkcji requestAnimationFrame do uzyskania efektu ciągłego wykonywania funkcji w celu odświeżania animacji.

3.3 Wzorce programowania asynchronicznego

Poważniejsze wyzwania asynchroniczne pojawiają się tam, gdzie pojawia się podsystem IO lub sieć. Funkcjonują dwa wzorce

3.3.1 Pojedyncze zdarzenie asynchroniczne

Jeśli obiekt „emituje” jedno asynchroniczne zdarzenie, node.js ma konwencję funkcji zwrotnej:

```
var fs = require('fs');

fs.readFile('a.txt', 'utf-8', function(err, data) {
    console.log( data );
});
```

3.3.2 Wiele zdarzeń asynchronicznych

Jeśli obiekt emituje wiele zdarzeń asynchronicznych, wzorzec funkcji zwrotnej nie sprawdza się. Przykładem byłby podsystem **http** w którym dane mogą spływać w wielu pakietach (jeden rodzaj zdarzenia), a po pewnym czasie następuje wyczerpanie transmisji (drugi rodzaj zdarzenia).

```
var http = require('https');

http.get('https://www.google.com', function(resp) {

    var buf = '';

    resp.on('data', function(data) {
        buf += data.toString();
    });

    resp.on('end', function() {
        console.log( buf );
    });
});
```

Dygresja: wzorzec w którym obiekt umożliwia obsługę wielu zdarzeń jest w inżynierii oprogramowania stosowany powszechnie, tu pod nazwą **EventEmiter** jest częścią biblioteki standardowej. Można emiterów używać wprost (lub dziedziczyć ich funkcjonalność przez ustawienie w łańcuchu prototypów jakiegoś obiektu), np.:

```
var EventEmiter = require('events');

var e = new EventEmiter();

e.on('start', function() {
    console.log('started');
});

e.on('work', function(payload) {
    console.log(`work: ${payload}`);
});

e.emit('start');
setTimeout( ()=> {
    e.emit('work', 17);
}, 1000);
```

3.4 Callback Hell

W obu podejściach kolejne wywołania funkcji asynchronicznych w jednym potoku powodują konieczność charakterystycznego zagnieżdżania funkcji zwrotnych, nazwanego żargonowo Callback Hell. Nieumiejętność radzenia sobie z tą niedogodnością struktury kodu jest jednym z powodów dla których Javascript miał przez lata tak niedobrą opinię.

```
var fs = require('fs');

fs.readFile('1.txt', 'utf-8', function(erra, dataa) {
    fs.readFile('2.txt', 'utf-8', function( errb, datab) {
        fs.readFile('3.txt', 'utf-8', function( errc, datac ) {
            console.log(dataa);
            console.log(datab);
            console.log(datac);
        });
    });
});
```

W powyższym przykładzie funkcji odczytującej kolejne pliki, ta struktura nie wygląda jeszcze źle, ale proszę na własną rękę zasymulować sytuację w której w pewnym miejscu kodu trzeba mieć wczytane dane z dwóch plików i odczytane zawartości z dwóch witryn internetowych.

Częściowym rozwiązaniem jest taka prosta refaktoryzacja kodu, w której kolejne zagnieżdżenia asynchronicznych funkcji zwrotnych stają się funkcjami na równorzędnym poziomie zagnieżdżenia

```
var fs = require('fs');

fs.readFile('1.txt', 'utf-8', function(erra, dataa) {
    readDataB(dataa);
});

function readDataB(dataa) {
    fs.readFile('2.txt', 'utf-8', function( errb, datab) {
        readDataC(dataa, datab);
    });
}

function readDataC(dataa, datab) {
    fs.readFile('3.txt', 'utf-8', function( errc, datac ) {
        console.log(dataa);
        console.log(datab);
        console.log(datac);
    });
}
```

W dłuższym kodzie nie jest to jednak wielki zysk.

3.5 Promise

3.5.1 Wprowadzenie

Alternatywą dla funkcji zwrotnych jest wzorzec struktury kodu oparty na obiektach Promise. Promise jest obiektem który przechowuje:

- Stan – może być
    - Pending – wyliczanie stanu trwa
    - Fulfilled – poprawnie wyliczono stan i jest on dostępny
    - Rejected – nie udało się wyliczenie stanu (odpowiednik wyrzucenia wyjątku)
- Funkcję do zmiany stanu, funkcja ta może być asynchroniczna (ale nie musi) – tę funkcję pisze programista i za jej pomocą (a konkretnie za pomocą funkcji zwrotnych przekazanych do tej funkcji) kontroluje przejście z Pending do Fulfilled albo Rejected
- **Listę** tzw. kontynuacji czyli funkcji, które trzeba wykonać wtedy kiedy wynik będzie dostępny (lub zostanie wyrzucony wyjątek) (ta lista może być pusta)

```
var p = new Promise( (res, rej) => {
    res(17);
});
```

```
p.then( result => {
    console.log( result );
})
```

Lub w wersji asynchronicznej

```
var p = new Promise( (res, rej) => {
    setTimeout( () => {
        res(17);
    }, 1000 );
});

p.then( result => {
    console.log( result );
})
```

Co ważne, dołączanie kontynuacji może nastąpić w dowolnym momencie, na przykład wtedy kiedy Promise jest już Fulfilled i ma już wartość

```
var p = new Promise( (res, rej) => {
    setTimeout( () => {
        console.log('ustawienie wartosci');
        res(17);
    }, 1000 );
});

setTimeout( () => {
    console.log( 'dodanie kontynuacji' );
    p.then( result => {
        console.log( result );
    });
}, 2000);
```

Konytnuacja z kolei zawsze zwraca Promise, nawet jeśli technicznie nie zwraca niczego (lub zwraca cokolwiek innego) – środowisko uruchomieniowe automatycznie przepisuje wtedy kod kontynuacji dodając zwrócenie Promise.

Dzięki temu możliwe jest **łańcuchowanie** wywołań:

```
var p = new Promise( (res, rej) => {
    res(17);
});

p
```

```
    .then( result => result + 1 )
    .then( result => {
        console.log( result );
    })
```

3.5.2 Zamiana Callback Hell na Promise

Czemu to wszystko służy? Pierwszym krokiem refaktoryzacji kodu asynchronicznego jest wprowadzenie Promise. Na przykład

```
var fs = require('fs');

function fspromise( path, enc ) {
    return new Promise( (res, rej) => {
        fs.readFile( path, enc, (err, data) => {
            if ( err )
                rej(err);
            res(data);
        });
    });
}

fspromise('a.txt', 'utf-8')
    .then( data => {
        console.log( `data: ${data}` );
    })
    .catch( err => {
        console.log( `err: ${err}` );
    })
```

Zamiast pisać własne funkcje zwracające **Promise**, będziemy posługiwać się funkcjami z biblioteki standardowej, tam gdzie są one dostępne. W przypadku modułu **fs**, mamy obiekt **fs.promises** a w nim dostęp do oryginalnych funkcji modułu fs tylko w wersjach zwracających Promise.

W pierwszej chwili może się wydawać, że przejście z callback na Promise nie rozwiązuje problemu „callback hell”, ponieważ naiwne stosowanie Promise powoduje identyczny efekt, jak ten którego chcemy unikać:

```
var fs = require('fs');

fs.promises.readFile('1.txt', 'utf-8')
.then(dataa => {
    fs.promises.readFile('2.txt', 'utf-8')
    .then( datab => {
        fs.promises.readFile('3.txt', 'utf-8')
        .then( datac => {
            console.log(dataa);
```

```
            console.log(datab);
            console.log(datac);
        })
    })
})
```

Tu jednak już prosta refaktoryzacja pomaga – pamiętając że **then** może zwrócić **Promise** do którego „przepinane" są kolejne **then.** Jedyna trudność techniczna polega na tym że z kontynuacji trzeba zwrócić zarówno wartość aktualną (którą już mamy) jak i „nową" (którą właśnie wyliczymy). Istnieje kilka technik poradzenia sobie z tym, najbardziej elganckie jest chyba użycie **Promise.all** które dla tablicy Promises zwraca Promise który zmienia stan dopiero wtedy kiedy zmienią stan wszystkie Promise z tablicy.

Tej funkcji używa się sprytnie – jeżeli jako argumenty dostanie na przykład konkretną wartość (już znaną) i oczekujące Promise, to tę znaną wartość zamienia na Promise i ostatecznie zwraca dwa obiekty Promise:

```
Var fs = require('fs');

fs.promises.readFile('1.txt', 'utf-8')
.then(dataa => {
    return Promise.all([dataa,fs.promises.readFile('2.txt', 'utf-8')]);
})
.then( ([dataa, datab]) => {
    return Promise.all([dataa, datab, fs.promises.readFile('3.txt', 'utf-
8')]);
})
.then( ([dataa, datab, datac]) => {
    console.log(dataa);
    console.log(datab);
    console.log(datac);
})
```

Promise.all umożliwia wręcz równoległe wywołanie obsługi IO – i tak, to dokładnie znaczy to co się wydaje, można podsystemowi IO zlecić odczytywanie dwóch (lub więcej) plików **naraz**:

```
var fs = require('fs');

function fspromise( path, enc ) {
    return new Promise( (res, rej) => {
        fs.readFile( path, enc, (err, data) => {
            if ( err )
                rej(err);
            res(data);
        });
```

```
    });
}

var f1 = fspromise('a.txt', 'utf-8');
var f2 = fspromise('b.txt', 'utf-8');

Promise.all( [f1,f2] )
    .then( ([a,b]) => {
        console.log(a,b);
    })
```

3.6 Async/await

Okazuje się, że wprowadzenie do języka obiektów Promise umożliwia dodanie warstwy lukru syntaktycznego, w której ciało kontynuacji jest włączone do ciała metody wywołującej Promise.

```
var fs = require('fs');

(async function () {

    var dataa = await fs.promises.readFile('1.txt', 'utf-8');
    var datab = await fs.promises.readFile('2.txt', 'utf-8');
    var datac = await fs.promises.readFile('3.txt', 'utf-8');

    console.log(dataa);
    console.log(datab);
    console.log(datac);

})();
```

Proszę uważnie przeanalizować co tu się dzieje – funkcja napisana jest syntaktycznie jako jedna funkcja. W rzeczywistości jednak po każdym **await** funkcja **kończy się** i wywołuje funkcję odczytującą plik, która po zakończeniu, do pętli zdarzeń wkłada nową funkcję, kontynuację (czyli to co jest napisane **po await**).

Ponieważ funkcja może zwracać kontynuacje dowolnie wiele razy (na przykład w nieskończonej pętli!), oznacza to w praktyce że taka funkcja oznakowana jako **async** technicznie zwraca z siebie … **generator** kolejnych kontynuacji, który po wywołaniu kolejnej funkcji może zwrócić następną itd.

Dla ciekawych – lukier syntaktyczny rozpisujący **async/await** ma więc wiele wspólnego z lukrem rozpisującym **yield**.

Dla czytelności kodu async/await ma niebagatelne znaczenie, dodatkowo – odpada obsługa klauzuli catch przez kontynuację, bo lukier syntaktyczny zamyka w **.catch** ciało bloku .. catch

```
```

```
async function main() {

    try {
        var a = await fspromise('a.txt', 'utf-8');
        var b = await fspromise('b.txt', 'utf-8');

        console.log(a,b);
    }
    catch ( e ) {

    }
}
```

Inny przykład, w którym w Promise opakowany jest obiekt emitujący wiele zdarzeń, przy czym szczęśliwie – jest wśród nich takie jedno (tu: **end**) które można potraktować jako wyzwalacz dla „Fulfilled”:

```
var http = require('http');

function promisedGet(url) {
    return new Promise(function (resolve, reject) {

    var client = http.get(url, function (res) {

        var buffer = '';

        res
            .on('data', function (data) {
                buffer += data.toString();
            })
            .on('end', function () {
                resolve(buffer);
            });
    });

    });
}

(async function() {

    var result = await promisedGet('http://www.google.pl');
    console.log( result );

})();
```

Bibilioteka standardowa coraz odważniej wprowadza funkcje zwracające **Promise** i należy się spodziewać że ten kierunek w przyszłości będzie dominującym stylem programowania asynchronicznego:

```
var fs = require('fs');

(async function() {

    var result = await fs.promises.readFile('a.txt', 'utf-8');
    console.log( result );

})();
```

3.7 Równoważność obu podejść

Podobnie jak na poprzednim wykładzie udało się nam pokazać równoważność obu technik obiektowych przez wyrażenie operatora **new** przez **Object.create** i **Object.create** przez **new**, tak tu sytuacja jest podobna – można napisać zarówno funkcję *promisyfikującą* wskazaną funkcję opartą o technikę *callback* oraz funkcję *depromisyfikującą* (?), która funkcję zwracającą *promise* zamieni na funkcję opartą o *callback*.

Ta równoważność ma duże znaczenie praktyczne, pokazuje bowiem że każdy z obu typów interfejsów programowania można przystosować do każdego sposobu korzystania z niego. W ten sposób można na przykład starsze API opakować tak żeby korzystać z niego w składni **async-await** albo odwrotnie – nowe API oparte o **async-await** przekazać do starszego fragmentu interfejsu spodziewającego się wywołań z funkcjami zwrotnymi.

W poniższym kodzie zdefiniowano wspomniane funkcje.

Funkcja **promisify** służy do zamiany funkcji spodziewającej się dodatkowego parametru (funkcji zwrotnej) na funkcję zwracającą **Promise**. Zwracana funkcja ma więc o jeden argument mniej niż pierwotna i do pierwotnej przekazuje wszystkie argumenty które otrzymuje oraz jeden, dodatkowy – funkcję zwrotną która w swoim ciele umożliwia zmianę stanu obiektu **Promise**.

Funkcja **unpromisify** służy z kolei do zamiany funkcji oryginalnie zwracającej **Promise** na taką funkcja ma o jeden argument więcej i tym argumentem jest funkcja zwrotna która przejmie na siebie obsługę asynchroniczności.

```
var fs = require('fs');

function promisify(f) {
    return function(...args) {
        return new Promise( (res,rej) => {
            f(...args, function(err,result) {
                if (err) {
                    rej(err);
                } else {
                    res(result);
                }
```

```
            });
        });
    }
}

function unpromisify(f) {
    return function(...args) {
        // args - od 0 do przedostatniego
        var params = args.slice(0,-1);
        // args - ostatni
        var cb     = args.slice(-1)[0];
        f(...params)
            .then( result => cb( null, result ) )
            .catch( err => cb(err) );
    }
}

(async function() {

    var _fsp = promisify( fs.readFile );
    var result = await _fsp('a.txt', 'utf-8');
    console.log( result );

    var _fsu = unpromisify( fs.promises.readFile );
    _fsu('a.txt', 'utf-8', function(err, data) {
        if ( err ) {
            console.log(err);
        } else {
            console.log( data );
        }
    });

})();
```