Západočeská univerzita v Plzni

Fakulta pedagogická

Katedra výpočetní a didaktické techniky

Nové možnosti ve standardu ECMAScript 6

BAKALÁŘSKÁ PRÁCE

Dominik Novosad

Informační technologie se zaměřením na vzdělávání

Vedoucí práce: PhDr. Tomáš Přibáň, Ph.D.

Plzeň 2019

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně s použitím uvedené literatury a zdrojů informací.

V Plzni, 26. dubna 2020

.....

vlastnoruční podpis

Chtěl bych poděkovat PhDr. Tomáši Přibáňovi, Ph.D. za odborné vedení práce a cenné rady, které mi pomohly tuto práci zkompletovat.

ZADÁNÍ

1. Představte ECMAScript v kontextu historického vývoje.
2. Uveďte, jaké nové možnosti nabízí standard ECMAScript 6.
3. Vytvořte sadu příkladů, ve kterých demonstrujete možnosti ECMAScriptu 6.

Obsah

SEZNAM ZKRATEK

ES – ECMAScript

JS - JavaScript

ÚVOD

JavaScript je skriptovací jazyk, jehož funkce využívá v dnešní době pravidelně i obyčejný člověk. Můžeme na něj narazit ve webových aplikacích, stránkách, nebo i serverech, dále i ve hrách, v grafickém designu a aplikacích pro chytré hodinky. Programování je rozhodně jednou z klíčových kompetencí budoucnosti vzhledem k průmyslu 4.0, s kterým úzce souvisí robotizace a automatizace. Jedná se o jazyk, který je objektově orientovaný a událostmi řízený. V praxi to znamená, že se metody zapouzdřují do objektů, což umožňuje snadný přenos kódu napříč několika projekty, a že je tok programu řízený událostmi od uživatelů jako např. stisk tlačítka, kliknutí a pohyb myši. Dále je JavaScript typický dynamickým přiřazováním datových typů. To znamená, že typy proměnných jsou dané daty, která do nich vložíme. Například pokud máme proměnnou X, můžeme jí přiřadit řetězec a později zase celočíselný typ. Existuje zde i nadstavba JavaScriptu zvaná TypeScript, která umožňuje pracovat se statickým typováním dat.

JavaScript je často zaměňován s programovacím jazykem Java, ale JavaScript je od tohoto jazyku značně odlišný, ať už jde o zápis kódu nebo jeho funkčnost. Slovo Java se v JavaScriptu vyskytuje pouze z marketingových důvodů. JavaScript byl krátce po jeho vytvoření standardizován společností ECMA, která novou standardizovanou verzi pojmenovala ECMAScript dále jen ES. ES od svého počátku prošel nespočtem aktualizací, jednou z nich je právě ES6, který přinesl obrovské množství změn a je to právě ten standard, který budu v této práci rozebírat.

1 ECMASCRIPT V KONTEXTU HISTORICKÉHO VÝVOJE

JavaScript podstoupil během svého vývoje takové množství změn jako málo programovacích jazyků před ním. Od přestavení Node js run-time v květnu roku 2009 se rozšířil dále nad rámec pouhých internetových prohlížečů. V současnosti se využívá na minipočítačích Raspberry Pi, jako skriptovací jazyk pro 3D počítačové hry na stolních počítačích, pro běh serverů, které načítají miliony zobrazených stránek denně a samozřejmě je to také dominantní jazyk pro webové prohlížeče. Je tedy možné, že JavaScript je nejdůležitějším programovacím jazykem na světě. (Harrison, 2018, s.1)

ES se zde pohybuje téměř stejně dlouho jako samotný JavaScript. Avšak během posledních pár let zažil mnoho změn. ES6 publikovaný v roce 2015, vytvořil téměř úplně nový programovací jazyk. Od té doby jsou aktualizace spíše menší, ale i přesto významné. (Harrison, 2018, s.1)

JavaScript byl vytvořen v roce 1955 Brendanem Eichem, který v té době pracoval v Netscape Communications Corporation. Brendan tehdy JavaScript pojmenoval Mocha, který byl krátce na to přejmenován na LiveScript a to před oficionálním přejmenováním na JavaScript. Tehdy se jednalo pouze o skriptovací jazyk, který běžel výhradně ve webovém prohlížeči, tedy na straně klienta. Používal se ve spojení s HTML k vytváření responsivních webových stránek.

(ES6 - Overview. © 2020.)

V roce 1995 si Netscape představoval dynamický web nad rámec toho, co tehdy mohlo HTML nabídnout. Brendan Eich měl původně v Netscapu vytvořit jazyk, který byl funkcionálně podobný Scheme, ale pro internetový prohlížeč. Jakmile na tom začal pracovat, tak zjistil, že management chtěl, aby jazyk vypadal jako Java. Brendan vytvořil první prototyp během deseti dní. Inspiraci si bral z funkcí ze Scheme a prototypy z jazyku Self. Původní verze JavaScriptu byla pojmenována Mocha. Neobsahovala žádná pole ani objekty a při každé chybě vyskočila varovná zpráva. Absence jakýchkoliv výjimek je důvod, proč do dnešního dne mnoho operací vede k „NaN“ nebo „undefined“. Brendanovo práce na základu DOM a první edice JavaScriptu postavila základy pro další práci na standardu. Vzhledem ke společné distribuci s Nescape Navigatorem 2.0 byla tato verze Javascriptu v roce 1995 po přezkoumání přejmenována na LiveScript. Později téhož

roku, když byla vydána další verze Navigatoru, byl LiveScript přejmenován z marketingových důvodů na JavaScript. Zanedlouho po tomto vydání Netscape představil JavaScriptovou implementaci na straně serveru pro skriptování v Netscape Enterprise Serveru a pojmenoval ji LiveWire1. (Bevacqua, 2016, s.12)

Microsoft se inspiroval JavaScriptem a vytvořil vlastní Jscript, který byl součástí Internet exploreru 3 v roce 1995. JScript byl dostupný pro Internet Information Server (IIS) na straně serveru. (Bevacqua, 2016, s.12)

Jazyk začal být v roce 1996 standardizován pod jménem ECMAScript(ES) dle specifikace ECMA-262 technologickým výborem známým jako TC39. Firma Sun nechtěla předat vlastnictví JavaScriptu společnosti ECMA, a ačkoliv Microsoft nabízel JScript, nechtěly ostatní společnosti toto jméno používat. Z tohoto důvodu se zůstalo u jména ECMAScript. (Bevacqua, 2016, s.13).

V této době spory kvůli konkurenčním implementacím, JavaScript od Netscapu a JScript od Microsoftu, dominovaly většině setkáním výboru TC39. Navzdory tomu všemu se výboru velmi dařilo. Už tehdy byla stanovena zpětná kompatibilita jako hlavní pravidlo pro budoucí práci. Byly zavedeny striktní operátory rovnosti (=== a !==) namísto toho, aby znemožnily funkce starším programům, které pracovaly s algoritmy bez těchto striktních operátorů. První edice ECMA-262 byla vydána v létě roku 1997. Rok poté byla specifikace zdokonalena tak, aby vyhovovala mezinárodním standardům ISO/IEC 16262 a byla formálně vydána jakožto druhá verze. V roce 1997 byla vydána třetí verze, která obsahovala regulární výrazy, switch příkazy, do/while, try/catch a Object#hasOwnProperty s mnoha dalšími změnami. Brzy poté byly zveřejněny návrhy na další verzi ES4 výborem TC39. Konfliktní názory na to, jak se má JavaScript v budoucnu posunout, kompletně zastavily jakýkoliv vývoj dalšího standardu. Jednalo se o velmi těžké období pro vývoj webových standardů. V této době Microsoft už téměř kompletně monopolizoval web a neměl téměř žádný zájem o vývoj dalších standardů. V roce 2003 AOL vyhodilo 50 zaměstnanců z Netscapu, načež byla založena nadace Mozilla. S 95% vlastnictvím webového trhu v rukou Microsoftu byl rozpuštěn výbor TC39. (Bevacqua, 2016, s.12)

Trvalo 2 roky než Brendan, který tou dobou pracoval v Mozille, znovu zahájil práci na TC39. Využil k tomu rostoucí akciový trh jakožto páku, která by dostala Microsoft do kouta.

V průběhu roku 2005 se TC39 začali pravidelně scházet a pracovat na nové verzi. ES4 měla obsahovat rozšíření jako systém modulů, třídy, iterátory, generátory, destrukturování objektů, anotace typů, koncové rekurze, algebraické datové typy a mnoho dalších prvků. Projekt byl natolik ambiciózní, že se práce na něm musela několikrát zastavit. V roce 2007 se výbor rozdělil na dvě poloviny: ES3.1, jenž měla v plánu postupně rozšiřovat ES3; a ES4, která byla redesignována a téměř nespecifikována. Až v roce 2008 se všichni společně rozhodli, že ES3.1 je ta správná cesta kupředu, ale formálně byla přejmenována na verzi ES5. Ačkoliv práce na ES4 byla zastavena, mnoho jejich plánovaných rozšíření se dostalo právě do ES6, která byla pojmenována Harmony (v době tohoto rozhodnutí), zatímco nad zbylými rozšířeními se pouze uvažovalo. Aktualizace ES3.1 sloužila jako pevný základ, ke kterému se postupně mohly přidávat jednotlivé malé elementy z ES4.(Bevacqua, 2016, s.13)

V prosinci 2009, u příležitosti desetiletého výročí od publikace ES3 byla vydána verze ES5. Tato verze v podstatě kodifikovala rozšíření specifikace jazyku, jenž se stala běžnými mezi implementacemi jednotlivých prohlížečů. V této verzi se nacházely nové implementace jako getter a settery, vylepšení funkcionality u prototypu polí, parsování formátu JSON a striktní mód. O pár let později v červnu 2011 byla verze znovu přezkoumána a upravena, aby se stala třetí verzí mezinárodního standardu ISO/IEC 16262:2011, a formalizována pod názvem ECMAScript 5.1. (Bevacqua, 2016, s.13).

TC39 trvalo další 4 roky než v červnu 2015 vydali ES6. Šestá verze byla největší aktualizací, jakou tento jazyk kdy zažil. Obsahovala mnoho aspektů z ES4, jenž byly odloženy kvůli práci na Harmony. (Bevacqua, 2016, s.13)

Současně s prací na ES6 v roce 2012 se společnost WHATWG(Web Hypertext Application Technology Working Group) rozhodla zdokumentovat rozdíly mezi ES5.1 a jednotlivými implementacemi prohlížečů. Za úkol bylo zjistit možné kompatibility a nároky na interoperabilitu. Tato skupina standardizovala String#substr, který doposud nebyl specifikován, sjednotila několik metod pro zapouzdřování řetězců do HTML elementu, jenž byly nekonzistentní napříč prohlížeči a zdokumentovala vlastnosti Object.prototype

jako např. „proto“, define Getter a provedla mnoho dalších vylepšení. Tato práce byla rozdělena napříč jednotlivými specifikacemi ECMAScriptu, které se později dostaly až do Annex B v roce 2015.(Bevacqua, 2016, s.13).

Annex B byla informativní sekce jádra specifikace ECMAScriptu, což znamenalo, že nebyly třeba další implementace pro dodržování jejich doporučení. Současně s tímto updatem se stal Annex B povinnou částí webových prohlížečů. (Bevacqua, 2016, s.14)

Šesté vydání bylo podstatným milníkem v historii JavaScriptu. Kromě tuctů nových funkcí, ES6 značí bod, kdy se ECMAScript stal primárním standardem pro vývoj nejen webových stránek.(Bevacqua, 2016, s.14)

JavaScript se vyvinul ze svých skromných začátků v roce 1995 na velmi impozantní jazyk, kterým je dnes. Ačkoliv je ES6 velkým krokem vpřed, rozhodně to není pro tento jazyk konec. Jelikož můžeme očekávat každý rok nové specifikace, je důležité ohledně aktuálních specifikací zůstat v obraze. Jednou z nejlepších metod, jak držet krok s vývojem je pravidelně navštěvovat stránky TC39 s návrhy pro další implementace ve 3. fázi, jenž mají největší pravděpodobnost se dostat do další verze.(Bevacqua, 2016, s.24)

2 NOVÉ MOŽNOSTI V STANDARDU ES6

Ve verzi ES6 se JavaScript rozšířil o nespočet nových funkcí. Ve verzi 5.1. měla specifikace jazyka zhruba 258 stránek, což se více než zdvojnásobilo na 566 stránek v ES6. Každá z nových funkcí se dělí mezi tyto kategorie: Syntactic sugar, nové mechaniky, lepší sémantika, nové metody a nová řešení pro existující limitace. Syntactic sugar je jedním z nejpodstatnějších elementů v ES6. Nová verze nabízí nové možnosti vyjadřování dědičnosti objektů, nové syntaxe tříd, používání zkrácených funkcí jako např. Arrow funkce a nové možnosti deklarování proměnných. (Bevacqua, 2016, s.23)

Mezi další funkce patří například destrukturování objektů, rest a spread, promises, které jsou novou možností, jak můžeme psát asychnronní kód. Dále iterátory, jenž reprezentují sekvenci hodnot, a generátory, jenž jsou speciálním druhem iterátorů, které mohou produkovat sekvence hodnot. Verze ES2017 rozšiřuje tyto možnosti s async/await, které nám dokonce umožnují psát asynchronní kódy, které se tváří jako synchronní. (Bevacqua, 2016, s.23)

2.1 LET A CONST

2.1.1 LET

Let je jednou z nejznámějších funkcí v ES6. Funguje podobně jak příkaz var, ale má jiná pravidla, co se týče jeho rozsahu. JavaScript měl vždy až příliš komplikovaná pravidla ohledně rozsahů, což přivádělo mnoho programátorů k šílenství, když se snažili nejprve zjistit, jak proměnné fungují v JavaScriptu. Jakmile narazíte na tzv. hoisting, což je vyzdvihování proměnných kdekoliv v kódu s rozsahem na celý kód jako například v kódu níže:

```
function isItTwo (value) {
if (value === 2) {
   var two = true
 }  return two
} isItTwo(2) // <- true
isItTwo('two') // <- undefined
```

Tento JavaScriptový kód funguje i přestože byla proměnná two deklarována uvnitř podmínky a poté se k ní přistupovalo z vně funkce. Díky tomu, jak je rozsah proměnných v JavaScriptu definován, je tento kód interpretován stejně, jako je popsáno v dalším kódu:

```
function isItTwo (value) {
  var two
if (value === 2) {
    two = true
}  return two
}
```

Ať už preferujeme jakoukoliv možnost, tak je zřetelné, že hoisting je mnohem více matoucí než používání proměnných v rozsahu jednoho bloku uvnitř složených závorek. (Bevacqua, 2016, s.49)

Namísto toho, abychom museli definovat nové funkce kdykoliv budeme chtít užší rozsah proměnných, rozsah pouze v bloku nám umožňuje využívat kódu ve větvích příkazů if, for, nebo while. Je také možné vytvářet vlastní bloky {} dle naší libosti. JavaScript umožňuje vytvářet neomezené množství bloků, pokud to vyžadujeme. (Bevacqua, 2016, s.49)

```
{{{{{ var deep = 'Tato proměnná je přístupná i z vně bloku.'; }}}}} console.log(deep) // <- ' Tato proměnná je přístupná i z vně bloku.'
```

K hodnotám přiřazeným ve var je možné se dostat i z vně bloku a bez chyby. Příkaz let je alternativou k var. S proměnnou var je jedinou možností jak získat užší rozsah deklarování vložených funkcí, ale s let stačí jen vytvořit blok kódu pomocí složených závorek. (Bevacqua, 2016, s.49)

```
let topmost = {}
{
let inner = {}
{
let innermost = {}
```

```
}  // Při pokusu o přístup k let innermost by zde vyskočila chyba

} // Při pokusu o přístup k let inner, innermost, by zde vyskočila chyba
```

Jedním z užitečných aspektů příkazů let je, že jdou použít při deklarování cyklů for a proměnné budou mít rozsah pouze na obsah cyklu, viz příklad níže:

```
for (let i = 0; i < 2; i++) {

 console.log(i)  // <- 0 // <- 1

} console.log(i) // <-  chyba (i is not defined)
```

(Bevacqua, 2016, s.49)

2.1.2 KONSTANTY

Proměnné deklarované použitím cons jsou považovány za konstanty. Jejich hodnoty nelze změnit, jakmile jsou definované. Z tohoto důvodu je třeba přiřadit const hodnotu už při její deklaraci jako v tomto příkladu:

```
const maxItems = 30; //správný zápis

const name; // Syntax error; chybí hodnota
```

(Zakas, 2016, p.4)

Konstanty fungují stejně jako let v blokovém rozsahu. Na následujícím příkladu můžeme vidět, že můžeme deklarovat konstantu se stejným jménem za předpokladu, že je v jiném bloku kódu.

```
const pi = 3.1415 {

const pi = 6

 console.log(pi)  // <- 6

 }
```

(Bevacqua, 2016, s.52)

Kromě vyžadování přirazené hodnoty při deklarování const, hodnotu těchto proměnných nelze ani měnit. Ve striktním režimu jakýkoliv pokus o změnu hodnoty vyvolá chybové

hlášení. Pokud kód proběhne v nestriktním režimu, tak selže bez chybového hlášení jako v následujícím příkladu:

```
const people = ['Tesla', 'Musk']

people = []

console.log(people) // <- ['Tesla', 'Musk']
```

(Bevacqua, 2016, s.53)

Použití const pouze znamená, že proměnná vždy bude mít referenci ke stejnému objektu či hodnotě, protože se daná reference nemůže měnit. Reference je sice neměnná, ale samotná hodnota uložená v proměnné se měnit může. Následující příklad názorně ukazuje, že i když reference na people nemůže být změněná, samotné pole může být modifikováno.

```
const people = ['Tesla', 'Musk']

people.push('Berners-Lee')

console.log(people) // <- ['Tesla', 'Musk', 'Berners-Lee']
```

(Bevacqua, 2016, s.54)

Příkaz const pouze proměnné brání v referenci jiné hodnoty. Další způsob, jak můžeme prokázat tento rozdíl je následující kód, kde vytvoříme proměnnou people za pomocí const a poté přiřadíme tuto hodnotu obyčejné proměnné var humans. Proměnné var poté můžeme přiřadit úplně jinou hodnotu, protože nebyla deklarována pomocí const, nicméně nemůžeme měnit hodnotu people, protože tato proměnná byla definována pomocí const.

```
const people = ['Tesla', 'Musk']

var humans = people

humans = 'evil'

console.log(humans) // <- 'evil'
```

(Bevacqua, 2016, s.54)

Pokud bychom chtěli mít proměnnou absolutně neměnnou, tak je třeba využít funkce jako je například Object.freeze, která brání jakémukoliv rozšiřování daného objektu, jak můžeme vidět v následujícím kódu.

```
const frozen = Object.freeze(['Ice', 'Icicle', 'Ice cube'])
frozen.push('Water') // Uncaught TypeError: Can't add property 3,
object is not extensible
```

(Bevacqua, 2016, s.54)

2.2 TEMPLATE LITERALS

Template literals jsou odpovědí ES6 na následující funkce, které v ES5 chyběly:

- Víceřádkové řetězce
- Základní formátování řetězce, možnost nahradit části řetězců za hodnoty obsažené v proměnných.
- HTML escaping – schopnost transformovat řetězce tak, aby šly bezpečně vložit do HTML. (Zakas, 2016, s.25)

Místo toho, aby se přidávala další funkcionalita k již existujícím řetězcům v JavaScriptu, template literals představují zcela novou možnost, jak řešit tuto problematiku. (Zakas, 2016, s.26)

Template literals pro svou syntax používají obrácené uvozovky (``) namísto běžných uvozovek jako u ostatních stringu. Template literal tedy může být zapsán takto:

```
let message = `Hello world!`;

console.log(message);   // "Hello world!"
```

(Zakas, 2016, s.26)

2.3 OBJECT LITERALS

Objectal literal je jedním z nejpoužívanějších vzorců v JavaScriptu. JSON je postaven na této syntaxi a existuje téměř v každém JavaScriptovém souboru na internetu. Object literal je velmi populární kvůli jednoduché syntaxi pro vytváření objektů, který by jinak zabraly několik řádek kódu. (Zakas, 2016, s.68)

2.3.1 PROPERTY INITIALIZER

V ECMAScriptu 5 a dříve byly object literals pouze soubory párů jméno-hodnota. To znamenalo, že se v kódu při inicializaci jednotlivých hodnot mohla vyskytnout duplikace. Například:

```
function createPerson(name, age){
return{
name: name,
age: age
};
}
```

(Zakas, 2016, s.68)

Funkce createPerson() vytvoří objekt, jehož proměnné mají stejné hodnoty jako parametry funkce. Ve výsledku to způsobí duplikaci name a age i přestože se v jednom případě jedná o proměnnou objektu a v druhém o hodnotu pro daný objekt. (Zakas, 2016, s.68)

V ES6 můžete eliminovat tuto duplikaci použitím zkratky property initializer. Když se jméno proměnné objektu shoduje s jménem lokální proměnné, stačí pouze toto jméno zapsat bez dvojtečky a přiřazení hodnoty jako v příkladu níže:

```
function createPerson(name, age) {
return{
name,
age
};
}
```

(Zakas, 2016, s.69)

2.4 DESTRUCTURING

Objekty a pole jsou běžně používanými datovými typy v JavScriptu a existuje mnoho postupů jak z nich systematicky získávat data. ES6 tento proces dále zdokonaluje a ulehčuje nám přístup k těmto datům procesem zvaným destrukturování.

(Grover, Kunduru, 2017, s.37)

Destrukturování je v podstatě zjednodušení dělení dat na menší díly, což nám umožňuje lepší přístup k datům a extrahování několika hodnot zároveň z objektů a polí.

(Grover, Kunduru, 2017, s.37)

```
var letters = ['a', 'b', 'c'],
x = letters[0],
y = letters[1],
z = letters[2];
console.log( x, y, z );  // a b c
```

(Grover, Kunduru, 2017, s.37)

V tomto příkladu jsme přiřadili hodnoty poli s názvem letters a poté proměnným x,y a z hodnoty z daného pole. Podívejme se na další příklad s objekty

(Grover, Kunduru, 2017, s.37)

```
var numbers = {a: 1, b: 2, c: 3},
 a = numbers.a,
 b = numbers.b,
 c = numbers.c;
console.log( a, b, c );  // 1 2 3
```

(Grover, Kunduru, 2017, s.37)

V tomto příkladu používáme hodnotu numbers.a k přiřazení hodnoty proměnné a podobně i u numbers.b a numbers.c pro proměnné b a c.

Všechny tyto procesy nám ES6 ulehčuje za pomoci destrukturování.

(Grover, Kunduru, 2017, s.37)

Tento zápis eliminuje potřebu pro deklarování proměnných jako letters a numbers.

```
var [ x, y, z ] = ['a', 'b', 'c'];

var { a: a, b: b, c: c } = {a: 1, b: 2, c: 3};

console.log( x, y, z );              // a b c

console.log( a, b, c );             // 1 2 3
```

(Grover, Kunduru, 2017, s.38)

Jak jsme mohli již vidět u předchozích dvou příkladů, před vydáním ES6, vyžadovalo získávání informací z objektů a polí za pomocí lokálních proměnných mnohem více kódu. Představte si, že potřebujete extrahovat hodnoty z komplexních objektů či polí a poté je musíte ukládat do proměnných se stejným jménem. Tímto způsobem musíte napsat mnohem více kódu postupným přiřazováním jednotlivých hodnot, ale díky destrukturování se tento proces může zjednodušit i na jednu řádku kódu.

(Grover, Kunduru, 2017, s.38)

2.4.1 DESTRUKTUROVÁNÍ OBJEKTŮ

Nyní se podívejme, jak můžeme využít destrukturování u objektů. Jak jsme již viděli, syntax pro destrukturování objektů vypadá takto:

```
var { a: a, b: b, c: c } = {a: 1, b: 2, c: 3};
```

(Grover, Kunduru, 2017, s.38)

V tomto příkladu si všimněte, že používáme stejné jméno pro proměnné a zároveň i pro hodnoty daného objektu. Jména nemusí být nutně stejná, pro lokální proměnné můžete zvolit jakékoliv jméno. Syntax může být ještě zkrácena vynecháním „a:“:

(Grover, Kunduru, 2017, s.39)

```
var { a, b, c } = {a: 1, b: 2, c: 3};

console.log( a, b, c );              // 1 2 3
```

(Grover, Kunduru, 2017, s.38)

Z hlediska správného psaní kódu pro nás dává smysl využívat kratší zápis, pokud tedy nechce přiřazovat hodnoty objektů proměnným s jiným jménem. Je zde ale jeden důležitý rozdíl, na který se podíváme v dalším příkladu a musíme si na něj dávat pozor.

(Grover, Kunduru, 2017, s.38)

```
var { a: foo, b: bar, c: baz } = {a: 1, b: 2, c: 3};

console.log( foo, bar, baz );          // 1 2 3

console.log( a, b, c );                // ReferenceError
```

(Grover, Kunduru, 2017, s.38)

Podívejme se podrobně na tento příklad. Zápis se může zdát na první pohled velmi jednoduchý, kde jsou levé straně přirazeny hodnoty z pravé strany { a: foo, b: bar, c: baz }, jak jsme zvyklí. Nicméně se jedná o úplný opak. Z jakého důvodu byl tedy zvolen tento zápis? Pojdmě se na to podívat podrobněji v následujícím příkladu.

(Grover, Kunduru, 2017, s.39)

```
var foo = 42, bar = 100;

var obj = { a: foo, b: bar };

 var { a: FOO, b: BAR } = obj;

console.log( FOO, BAR );               // 42 100
```

(Grover, Kunduru, 2017, s.39)

V tomto příkladu, proměnné a a b obsahují hodnoty objektu a v řádce kódu s destrukturováním, a a b také reprezentují data z objektu. Pamatujete, jak jsme v předchozím příkladu zmínili, že můžeme zkrátit kód vynecháním „a:“. Je to přesně z tohoto důvodu. Pokud v tomto příkladu vynecháme část syntaxe a: a b:, zůstane nám pouze FOO a BAR. Tato syntax se může zdát matoucí, nicméně je velmi jednoduchá na pochopení, stačí si uvědomit, že jména daných property a jejich hodnoty jsou k sobě připojeny za předpokladu, že se stejně jmenují. V tomto případě se foo přiřadí k FOO a bar k BAR použitím jmen pro property a a b. Za pomocí destrukturování objektů můžete i přiřadit hodnoty několika proměnných použitím stejné hodnoty property objektu. Podíváme se, jak to udělat v následujícím příkladu:

(Grover, Kunduru, 2017, s.39)

```
var { x: foo, x: bar } = { x: 42 };

console.log( foo );  // 42 console.log( bar );  // 42
```

(Grover, Kunduru, 2017, s.39)

V tomto případě foo a bar získají jejich hodnotu za pomocí property objektu x. Můžete používat i var, let, a const dle vaší situace a požadavků při destrukturování. Nesmíte ale nikdy vynechat přiřazení hodnoty, tedy pravou stranu daného výrazu.

(Grover, Kunduru, 2017, s.39)

```
var { x, y };      // syntax error! let

{ x, y };         // syntax error! const

{ x, y };       // syntax error!
```

(Grover, Kunduru, 2017, s.39)

Všechny tyto deklarace způsobí syntax error, protože jim chybí inicializace v deklaraci destrukturování.

(Grover, Kunduru, 2017, s.39)

2.5 DEFINOVÁNÍ TŘÍD

Třídy byly vždy velmi debatovaným tématem při vývojovém procesu ES6. Vývojáři nakonec došli ke kompromisu. Je důležité pochopit, že třídy v ES6 nefungují úplně stejně jako v ostatních objektově orientovaných jazycích. TC39 je pořád v procesu přidávání dalších funkcí tříd po ES6 tak, aby se více přiblížily ke klasickým třídám. Co se týče konceptu jako takového, v tradičním JavaScriptu třídy neexistují. Třída v ES6 je pouze funkce, která se jako třída tváří. Třídy v ES6 podporují dědičnost založenou na prototypech, konstruktory, super, instance, a statické metody. Podívejme se na to, jak můžeme definovat základní třídu, což je velmi podobné tomu, jak se to dělá v jiných jazycích. (Grover, Kunduru, 2017, s.49)

```
class Car {

constructor(brand) {
```

```
 this.brand = brand;
   }
}
const myTesla = new Car("Tesla");
console.log (myTesla.hasOwnProperty("brand"));  // true
console.log (typeof Car);                  // function
```

V tomto příkladu je brand vlastnost přiřazená objektu myTesla. Tento objekt může být také zapsán pomocí object literal takto:

```
const myTesla = { brand: "Tesla" };
```

Výše zmíněný příklad je jednoduchá reprezentace, jak vytvořit třídu s fakultativním konstruktorem, který bude zavolán, když je vytvořena instance objektu této třídy. Třídy v ES6 nepřidávají žádnou novou funkcionalitu k tomu, co už se momentálně v jazyku nachází. Jedná se pouze o zjednodušenou syntax pro pracování s objekty. Jakýkoliv nový argument, který přidáme do new Car() bude přijat jako parametr pro metodu konstruktoru Car a tyto parametry můžete použít pro vytvoření instance stejně třídy. (Grover, Kunduru, 2017, s.50)

Constructor() je tedy metodou, kde inicializujeme vlažnosti objektu a jelikož je constructor() fakultativní, můžeme založit prázdnou třídu jako v následujícím příkladu:

(Grover, Kunduru, 2017, s.50)

```
class EmptyClass {
}
```

Pokud nedefinujete konstruktor uvnitř třídy, JavaScript tam vloží prázdný za Vás.

```
class EmptyClass {
    // JavaScript vloží prázdny konstruktor:     constructor () {
}    */
}
```

2.5.1 DEKLARACE METOD UVNITŘ TŘÍD

Deklarování metod uvnitř tříd je velmi podobné vytváření klasických funkcí. Rozdíl je v tom, že zde není třeba používat klíčové slovo function. (Grover, Kunduru, 2017, s.50)

```
class Car {
constructor(brand) {
 this.brand = brand;
}
 start() {
 console.log(`Your ${this.brand} is ready to go!`);        } }
const myTesla = new Car("Tesla");
myTesla.start(); // Your Tesla is ready to go!
```

V tomto příkladu jsme definovali metodu start() uvnitř třídy, která pracuje s vlastností brand za pomoci klíčového slova this. (Grover, Kunduru, 2017, s.51)

2.6 MODULY

Na svém počátku začínal JavaScript jako jednoduchý skriptovací jazyk pro webové prohlížeče. Z toho důvodu byl původně koncipován tak, že v podstatě všechno bylo sdíleno. Ale jak plynul čas a JavaScript se stával mnohem komplexnější, se tento přístup k načítání kódu stával mnohem víc matoucí s větší pravděpodobností na vznik chyb. Před vydáním ES6 se všechno nacházelo uvnitř JavaScriptové aplikace, včetně kódu napříč různými soubory aplikace, sdílelo stejný rozsah neboli scope. Zde přichází na pomoc moduly, které nám umožňují lépe strukturovat kód, snižují množství konfliktů z důvodů kolizí jmen u proměnných a celkově zvyšují zabezpečení dat. Před nástupem ES6 museli vývojáři využívat externích knihoven, které podporovaly moduly, aby se těmto problémům vyhnuli. (Grover, Kunduru, 2017, s.65)

Moduly v ES6 nám umožňují lépe rozčlenit kód do menších úryvků, které poté můžeme tnovy využít a vložit(importovat) do jiných míst, kde jsou vyžadovány. Toto ulehčuje i testování kódu, jelikož jsou moduly odděleny od vašeho primárního kódu. Moduly jsou také užitečné k asynchronnímu načítání kódu, což výrazně zrychluje načítání applikace.

Systémy modulů ale nejsou žádnou novinkou v JavaScriptu. Moduly jsou pouhé JavaScriptové soubory, které načteme do JavaScriptového kódu z jiného souboru. Je ale důležité pochopit rozdíl mezi moduly a klasickým importováním skriptů. V první řadě moduly vždy běží ve striktním režimu na rozdíl od tradičních skriptů, kde to můžeme změnit. Pokud se podíváme na vrchní úroveň rozsahu(scope) modulu, tak uvidíme, že hodnota this je undefined a proměnné deklarované ve vrchní úrovni nejsou v globálním rozsahu viditelné. Cokoliv, co potřebujete zevnitř modulů, musí být nejdříve explicitně exportováno, abychom to mohli použít vně modulu. (Grover, Kunduru, 2017, s.66)

Příkaz export lze použít pro odhalení částí kódu ostatním modulům. Exportovat můžete proměnné, funkce, nebo samotné třídy. Proměnné, funkce a třídy neexportované z modulu nejsou dostupné mimo modul, ve kterém se nacházejí. (Grover, Kunduru, 2017, s.67)

```
export var text = "ES6 is awesome";

export let name = "Ian Murawski";

export const number = 7;
```

V tomto příkladu exportujeme text, name, a number. Všechny tyto proměnné jsou deklarovány jiným způsobem. (Grover, Kunduru, 2017, s.67)

```
export function add(a, b) {
    return a + b;
}

export class Rectangle {
   constructor(length, width) {
    this.length = length;
    this.width = width;
 }
 }
```

V tomto úryvku kódu exportujeme funkce add a třídu Rectangle. Můžeme také exportovat existující funkci, která je v modulu privátní.

```
function multiply(a, b) {
   return a * b;
}
 export { multiply };
```

V tomto příkladu můžeme vidět, že mohou být exportovány nejenom deklarace funkce, ale i reference na danou funkci.

V Node.js je běžnou praktikou používat moduly, které exportují pouze jednu hodnotu. I při používání JavaScriptu pro front-end, kde používáme třídy pro jednotlivé komponenty, je jedna třída pro modul zcela běžná. (Grover, Kunduru, 2017, s.67)

Jedna proměnná, funkce, či třída může být specifikována jako defaultní export modulu za použití klíčového slova export. Můžete mít pouze jednu hodnotu jako defaultní export uvnitř modulu. Používání exportu více než jednou uvnitř modulu způsobí chybu. (Grover, Kunduru, 2017, s.68)

```
export default function(a, b) {
    return a * b;
 }
```

V tomto příkladu je funkce exportována z modulu jakožto jeho defaultní. Můžeme si všimnout, že funkce nevyžaduje žádné jméno, jelikož je defaultní a modul sám o sobě reprezentuje danou funkci, což nám umožňuje jméno vynechat. (Grover, Kunduru, 2017, s.68)

Podobně jako v předešlých příkladech, můžeme exportovat referenci na funkci takto:

```
function multiply(a, b) {
   return a * b;
}
export default multiply;
```

(Grover, Kunduru, 2017, s.68)

Jakmile máme modul naexportovaný, můžeme k němu přistupovat z jiného modulu za pomocí klíčového slova import. Příkaz import má dvě části: identifikátor, který

importujeme a modul z kterého tento identifikátor importujeme. Toto je příkaz ve své základní formě: (Grover, Kunduru, 2017, s.68)

```
import { identifier1, identifier2 } from "./moduleFile.js";
```

Ve výše zmíněném příkaze, identifier1 a identifier2 jsou vazby importované z moduleFile.js. Modul je specifikován za použití řetězce, který ukazuje cestu k souboru obsahující module po slově „from“. Tyto vazby jsou definované pomocí const, tudíž nelze definovat žádné jiné proměnné se stejným jménem, nebo importovat další modul taktéž se stejným jménem. (Grover, Kunduru, 2017, s.68)

```
// importing the functions sum and multiply
import { sum, multiply } from "./ moduleFile.js";
 console.log(sum(1, 7));                // 8
 console.log(multiply(2, 3));           // 6
```

Zde jsou importovány dvě vazby z modulu moduleFile.js. Sum a multiply mohou být použity jako jakákoliv lokálně deklarovaná metoda. Pokud chceme importovat všechny dostupné exporty z modulu, musíme explicitně použít značku „*“.(Grover, Kunduru, 2017, s.68)

```
// importování všeho
import * as example from "./ moduleFile.js";
console.log(example.sum(1,7));      //8
console.log(example.multiply(2, 3));    // 6
```

V tomto kódu jsou importovány všechny exporty v modulu moduleFile do objektu s názvem example. Všechny exporty modulu jsou nyní dostupné jakožto vlastnost deklarovaného objektu. Takto vznikne objekt s novým jménem, jelikož tento objekt neexistuje v daném modulu. (Grover, Kunduru, 2017, s.69)

2.7 ARROW FUNKCE

Jedním z nejzajímavějších aspektů ES6 jsou arrow funkce. Arrow funkce jsou, jak už jméno napovídá, funkce definované novou syntaxí, která používá „šipku“ (=>). Nejedná se pouze o jiný zápis, arrow funkce se od těch tradičních liší ve funkcionalitě v několika ohledech.

(Zakas, 2016, s.54)

- Funkce nemůže být zavolána pomocí new – Arrow funkce nemají [[Construct]] metodu, tudíž nemohou být použity jako kontruktory. Při zavolání Arrow funkce pomocí new vyskočí chybové hlášení.
- Žádný prototype – Jelikož nelze použít na funkci new, tak není žádná potřeba pro Prototype.
- Nelze změnit this – Hodnota this nelze změnit uvnitř funkce. Hodnota zůstává stejnou po životní cyklus funkce.
- Žádné argumenty objektů – Jelikož arrow funkce nemají žádné přiřazování argumentů, tak je třeba spoléhat na namet a rest parametry pro přístup k argumentům funkcí.
- Žádné parametry s duplicitním jménem – Arrow funkce nemůžou mít duplicitní jména u parametrů ve striktním i nestriktním režimu. Na rozdíl od běžných funkcí, kde toto pravidlo platí pouze ve striktním režimu. (Zakas, 2016, s.55)

Existuje hned několik důvodů pro tyto rozdíly. V první řadě je změna hodnoty this častým zdrojem chyb v JavaScriptu. Je velmi jednoduché ztratit přehled o hodnotě this uvnitř funkce, což může vést k nezamyšlenému běhu program a arrow funkce tento problém eliminují. Druhým důvodem je, že omezením arrow funkce na používání jednotné hodnoty this dokáže engine JavaScriptu lépe optimalizovat tyto procesy, a to na rozdíl od obyčejných funkci, které mohou být použity jako konstruktory, nebo mohou být jinak modifikovány.

Zbylé rozdíly se soustředí na zmenšování množství chyb a nejasností uvnitř arrow funkcí. Díky tomu může engine JavaScriptu lépe optimalizovat provedení arrow funkcí. (Zakas, 2016, s.55)

2.7.1 SYNTAX ARROW FUNKCE

Syntax arrow funkce se může lišit podle toho, čeho chceme s danou funkcí dosáhnout. Všechny variace začínají s argumenty funkce, za nimi následuje =>, a poté už jen samotné tělo funkce. Například následující funkce má pouze jeden argument, který pouze vrátí.

```
let reflect = value => value;
```

//v porovnání s ES5 zápisem:

```
let reflect = function(value) {    return value; };
```

Pokud má funkce pouze jeden argument, tak může být zapsán bez jakékoliv další syntaxe. I přestože zde nevolá explicitně return, tak funkce vrací první argument, který dostane. (Zakas, 2016, s.55)

Pokud předáváte funkci více než jeden argument, je třeba argument umístit do závorek takto:

```
let sum = (num1, num2) => num1 + num2;
```

// v porovnání s ES5 zápisem:

```
let sum = function(num1, num2) {    return num1 + num2; };
```

Funkce sum() pouze sečte dva argumenty dohromady a vrátí výsledek. Jediný rozdíl mezi touto funkcí a funkci reflect je, že jsou argumenty uzavřeny v závorkách a odděleny čárkou. Pokud funkce nemá žádný argument, tak je třeba do deklarace umístit prázdné závorky takto:

```
let getName = () => "Nicholas";
```

// v porovnání s ES5 zápisem:

```
let getName = function() {    return "Nicholas"; };
```

(Zakas, 2016, s.55)

Pokud chcete vytvořit funkci se spíše tradičním obsahem, ve kterém se nachází více než jeden výraz, je třeba zabalit tělo funkce do složených závorek jako v následující funkci sum():

```
let sum = (num1, num2) => {    return num1 + num2; };
```

// v porovnání s ES5 zápisem:

```
let sum = function(num1, num2) {    return num1 + num2; };
```

(Zakas, 2016, s.55)

2.8 PROMISES

V předchozích verzích JavaScriptu byl callback nejběžnější způsob, jak organizovat asynchronní kód. Funkci to sice splňovalo, nicméně to způsobovalo určité komplikace. Čím více asynchronních funkcí se s callbacky přidalo, tím hůře se v kódu orientovalo a bylo mnohem těžší ho i upravovat. Tato situace byla velmi často nazývaná jako „callback hell".(Harrison, 2018, s.127)

Asynchronní kód je takový kód, který se neprovádí okamžitě, ale po určitém čase, nebo někdy v budoucnu. Callback je obecný název pro funkce, které vrací výsledek až po nějakém čase. (Callback Hell)

Promises byly vytvořeny, aby zlepšily tuto situaci. Promises nám umožňují měnit a lépe organizovat vztahy u asynchronních operací s větší svobodou a flexibilitou. (Harrison, 2018, s.127)

2.8.1 CYKLY U PROMISES

Každý promise má svůj průběh fungovaní o několika fázích. Hned první je fáze pending, která indikuje, že asynchronní operace ještě nebyla provedena. Jakmile je asynchronní operace dokončena, tak je považován promise za splněný a vstoupí do jednoho ze dvou možných stavů:

- 1. Fulfilled: Asynchronní operace byla úspěšně provedena.
- 2. Rejected: Asynchronní operace nebyla úspěšně provedena z důvodu chyby či jiné komplikace.

(Zakas, 2016, s.217)

Vnitřní vlastnost [[Promise state]] je přiřazena hodnota pending, fulfilled nebo rejected, aby odrážela stav daného promise. Tato vlastnost není viditelná v objektu promise, tudíž není možné zjistit ve kterém stavu se promise programově nachází. Je ale možné provést určitou funkci v momentě, kdy promise změní svůj stav pomocí metody then().(Zakas, 2016, s.217)

Metoda then() je přítomná ve všech promises a vyžaduje dva argumenty. První je funkce, která se má zavolat po splnění promise. Jakákoliv dodatečná data spojená s asynchronní operací jsou poslána právě této funkci. Druhým argumentem je funkce, která se má zavolat, pokud promise spadne do stavu rejected. Stejně jako v prvním případě, jakákoliv data spojená se selháním jsou poslána této funkci. (Zakas, 2016, s.217)

2.8.2 SYNTAX U PROMISES

```
Const myPromise= new Promise((resolve, reject) =>{
      if(Math.random() * 100 <= 90) {
      resolve(„Hello, Promises!“);
      }
reject(new Error(„ V 10% případech selžu“);
})
const myPromise = new Promise((resolve, reject) => {
if(Math.random() * 100 < 90) {
console.log („resolving promise“);
resolve(„Hello, promises“);
}
Reject(new Error(„V 10% případů selžu“);
// dvě funkce
const onResolved = (resolvedValue) =>
console.log(resolvedValue);
const onRejected = (error) => console.log(error);
myPromise.then(onResolved, onRejected);
// To samé jako výše zmíněný kód, ale stručnější
myPromise.then((resolvedValue) => {
console.log(error);
});
```

```
// výstup v 90% případů

// resolving promise

// Hello promises

//Hello promises
```

2.8.3 METODA PROMISE.ALL()

Metoda Promise.all() přijme jeden argument ve formě pole, ve kterém se nachází seznam promises, které bude tato metoda monitorovat. Tato metoda vrací promise, který je resolved pouze v případě, když jsou resolved všechny promises, které jsme určili.

(Zakas, 2016, s. 234)

```
let p1 = new Promise(function(resolve, reject) {
resolve(42);
});
let p2 = new Promise(function(resolve, reject) {
resolve(43);
});
let p3 = new Promise(function(resolve, reject) {
resolve(44);
});
let p4 = Promise.all([p1, p2, p3]);
p4.then(function(value) {
console.log(Array.isArray(value)); // true
console.log(value[0]); // 42
console.log(value[1]); // 43
console.log(value[2]); // 44
});
```

(Zakas, 2016, s. 234)

V tomto kódu je každý promise resolved číslem. Zavoláním metody Promise.all() vytvoříme promise p4, který je splněn pouze za předpokladu, že jsou splněny promises p1,p2 a p3. Výsledkem promise p4 je pole všech resolved hodnot, tedy: 42,43, a 44. Tyto hodnoty jsou uloženy v pořadí ve kterém jsou jednotlivé promises resolved. Pokud jakýkoliv z promises poslaný metodě Promise.all() je rejected, tak je okamžitě rejected i návratový promise metody bez čekání na to, jak dopadnou ostatní promises.

(Zakas, 2016, s. 234)

3 SADA PŘÍKLADŮ

V této kapitole představím sadu příkladů na na nové možnosti, které jsem představil v teoretické části. Veškerá řešení jsou dostupná na:

https://codepen.io/ECMAScript6

3.1 LET A CONST

3.1.1 ZÁKLADNÍ PRINCIPY LET

Požadované dovednosti

- znalost let
- povědomí o tom, jak funguje scope

Cíl

Seznámit se ze základními principy let.

Zadání

V úryvku níže naleznete jednoduchý kód využívající let, který na základě podmínky vypíše text do konzole. Nicméně takto napsaný kód způsobí chybu „ReferenceError: hello is not defined“. Upravte kód tak, aby správně vypsal výsledek do konzole.

Kód

```
var name = "Peter";
if(name === "Peter"){
  let hello = "Hello Peter";
} else {
  let hello = "Hi";
}
console.log(hello);
```

Řešení

```
var name = "Peter";
if(name === "Peter"){
  let hello = "Hello Peter";
} else {
  let hello = "Hi";
```

```
console.log(hello);

}
```

Zdroj: Kód převzatý z: https://www.sitepoint.com/how-to-declare-variables-javascript/

Závěr

Jelikož let má scope pouze v rámci bloku, ve kterém je umístěn, nelze se na něj odkazovat z vně daného bloku, proto se tento problém dá vyřešit pouhým přesunutím console.log do daného bloku.

3.1.2 POUŽITÍ LET A CONST

Požadované znalosti

- Znalost let a const

Cíl

Pochopit, kde používat let a const.

Zadání

Analyzujte níže přiložený kód, který spočítá množství číslic v řetězci a rozhodněte, kde bude vhodné nahradit var za let a const.

Kód

```
function spocitej(vstupniRetezec){
    var znaky = ['1','2','3','4','5','6','7','8','9','0',];
    var pocet = 0;

    for(var i = 0; i < vstupniRetezec.length; i++ ){
          if(znaky.includes(vstupniRetezec[i])){
           pocet++;
          }
    }
    return pocet;
}
console.log(spocitej("af1d5jksfdků2js4dfjků"));
```

Zdroj: Vlastní kód

Řešení

```
function spocitej(vstupniRetezec){
    const znaky = ['1','2','3','4','5','6','7','8','9','0',];
```

```
    let pocet = 0;

    for(let i = 0; i < vstupniRetezec.length; i++ ){
          if(znaky.includes(vstupniRetezec[i])){
           pocet++;
          }
    }
    return pocet;
}
console.log(spocitej("af1d5jksfdků2js4dfjků"));
```

Zdroj: Vlastní kód

Závěr

Jelikož víme, že tato funkce ověřuje výskyt číslic v řetězci za pomocí neměnného řetězce znaky, tak zde můžeme použít const. Na druhou stranu proměnná pro počet číslíc se zvětší pří každém výskytu čísla v řetězci, proto je zde vhodné použít let. Správným použitím let a const můžeme nejen zabránit vzniku chyb při rozšiřování daného kódu, ale také zřetelně ukazujeme, co se s danými proměnnými v kódu bude dít, tudíž je kód i mnohem zřetelnější pro někoho, kdo ho nepsal.

3.2 TEMPLATE LITERALS

3.2.1 ZÁKLADY TEMPLATE LITERALS

Úvod

Template literals je jedním z nových prvků, který patří do kategorie syntaktického cukru. Pomáhají nám výrazně zkrátit a zpřehlednit zápis řetězců v našem kódu tím, že nahradí velmi nepřehledné spojování řetězců pomocí +, které je často velmi matoucí.

Požadované dovednosti

- znalost syntaxe template literals

Cíl

Pochopit, kde používat let a const.

Zadání

Níže můžete vidět funkci, která po zavolání vypíše údaje o studentovi. Zjednodušte kód za použití template literals.

Kód

```
function vypisStudenta(jmeno, vek, obor, titul){
    return "Jmeno: " + jmeno + ", vek: " + vek + ", obor: " + obor + ", titu
l: " + titul

  }
  console.log(vypisStudenta("Dominik Novosad", "24", "IT", " "));
```

Zdroj: Vlastní kód

Řešení

```
function vypisStudenta(jmeno, vek, obor, titul){
    return `Jmeno: ${jmeno} , vek:${vek} , obor: ${obor}, titul: ${titul}`;

  }
  console.log(vypisStudenta("Dominik Novosad", "24", "IT", " "));
```

Zdroj: Vlastní kód

Závěr

Při použití template literals je třeba řetězec uzavřít do „`“ a proměnné vložit do složených závorek předcházející „$“. Ve složených závorkách můžeme volat i metody, nebo dále modifikovat proměnné. Například pokud bychom místo věku chtěli získat datum narození, můžeme to udělat takto:

```
  function vypisStudenta(jmeno, vek, obor, titul){
    return `Jmeno: ${jmeno} , vek:${new Date().getFullYear() -
 vek} , obor: ${obor}, titul: ${titul}`;

  }
```

Zdroj: Vlastní kód

3.3 OBJECT LITERALS

3.3.1 KDE POUŽÍT OBJECT LITERALS

Úvod

Object literals spadá do kategorie syntaktického cukru, tudíž nám toho nepřináší moc z pohledu nových funkcionalit, nicméně nám umožňuje zkrátit a zjednodušit náš kód. V tomto příkladu se podíváme na to, jak toho můžeme využít.

Požadované dovednosti

- znalost syntaxe object literals

Cíl

Pochopit kde používat object literals

Zadání

Níže naleznete kód, který na základě daného řetězce s předměty rozhodne, zda máte dostatek kreditů pro ukončení bakalářského studia. Využijte object literals ke zjednodušení funkce vytvorDatabaziKreditu.

Kód

```
function vytvorDatabaziKreditu(predmety) {
    return {
      predmety: predmety,
      pocetKreditu: function () {
        return this.predmety.reduce((soucet, predmet) => soucet + predmet.kr
edity, 0);
      },
    };
  }function overMnozstviKreditu() {
    if (databazeKreditu.pocetKreditu() >= 180) {
      console.log("Máte dostatek kreditů");
    } else {
      console.log("Zapište si další předměty");
    }
  }const predmety = [
    { jmeno: "Programovani 1", kredity: 4 },
    { jmeno: "Webove technologie", kredity: 3 },
    { jmeno: "Programovani 2", kredity: 5 },
    { jmeno: "DevOps", kredity: 6 },
```

```
  ];
  const databazeKreditu = vytvorDatabaziKreditu(predmety);
  overMnozstviKreditu();
```

Zdroj: Vlastní kód

Řešení

```
function vytvorDatabaziKreditu(predmety) {
    return {
      predmety,
      pocetKreditu() {
        return this.predmety.reduce((soucet, predmet) => soucet + predmet.kr
edity, 0);
      },
    };
  }
```

Zdroj: Vlastní kód

Závěr

Object literals nám umožňují obejít zdlouhavé přiřazování hodnot u objektů tím, že pokud přiřazujeme property objektu parametr s totožným jménem, tak stačí toto jméno uvést pouze jednou. U rozsáhlejších kódů, kde se tento proces několikrát opakuje, můžeme ušetřit mnoho místa a kód výrazně zkrátit. Zároveň díky object literals není třeba uvádět, že daná hodnota je funkce za pomocí dvojtečky a slova function.

3.4 DESTRUCTURING

3.4.1 DESTRUCTURING POLE VE FUNKCI

Vyžadované dovednosti

- Znalost Destrukturování
- Schopnost pracovat s knihovnou date
- Základní znalosti JS jako funkce

Cíl

Naučit se používat destructuring s poli.

Zadání

Vytvořte funkci, která vypočítá rok odchodu do důchodu se vstupním parametrem roku narození dané osoby. Funkce bude na konci vracet objekt s parametry věku osoby, a počet let za jak dlouho půjde do důchodu. Tento objekt poté uložte pomocí destructuring a vypište na konzoli. Jako pomůcku pro zjištění aktuálního roku můžete využít knihovny Date.

```
function vypocitejRokDuchodu(rok) {
  const vek = new Date().getFullYear() - rok;
  return [vek, 65 - vek];
}

const [vek2, duchod] = vypocitejRokDuchodu(1996);
console.log("Věk: ", vek2);
console.log("Důchod: ", duchod);
```

Zdroj: Vlastní kód

3.4.2 DESTRUCTURING V ZÁPISU FUNKCÍ

Níže můžete vidět funkci pro vytvoření nového uživatele s několika argumenty a později volání této funkce. Problém v tomto zápisu je, že pokud bychom chtěli například volat tuto funkci z jiného souboru nebo z místa, kde nevidíme na deklaraci dané funkce, tak se musíme vrátit k deklaraci funkce a opsat všechny argumenty a ujistit se, že je do zápisu volání funkce dáváme ve správném pořadí.

Vyžadované dovednosti

- Znalost Destrukturování
- Základní znalosti JS jako funkce, deklarování objektů

Cíl

Naučit se používat destructuring ve funkcích.

Zadání

Zjednodušte zápis tohoto kódu pomocí destructuring.

Nápověda

Zkuste při volání funkce argumenty poslat jako objekt.

Kód

```
 function registrace(uzivatelskeJmeno, heslo, email, datumNarozeni, mesto){

        //Vytvoření nového uživatele
      }
      registrace("Dominik123","qwertz","email@seznam.cz", "13/02/1996","Tach
ov");
```

Zdroj: Vlastní kód

Řešení

```
function registrace({uzivatelskeJmeno, heslo, email, datumNarozeni, město}){
    //Vytvoření nového uživatele
    }
    const uzivatel = {
    uzivatelskeJmeno: “Dominik123”,
    heslo: “qwertz”,
    email: “email@seznam.cz“,
    datumNarozeni: „13/02/1996,
    město: „Tachov“¨
    }
    registrace(uzivatel);
```

Zdroj: Vlastní kód

Závěr

Destructuring nám v tomto kódu zjednodušuje budoucí volání funkcí tím, že při každém volání stačí poslat funkci jenom daný objekt, zároveň nám to umožňuje volně pohybovat s pořadím jednotlivých argumentů ve funkci.

3.4.3 DESTRUCTURING ŘETĚZCE OBJEKTŮ

Požadovné dovednosti

- Znalost syntaxe object literals

Cíl

Naučit se používat destructuring s objekty.

Zadání

Za pomocí destructuring získejte hodnotu kraje v prvním objektu v řetězci.

Řešení

```
const Mesta = [
  { jmeno: "Tachov", kraj: "Plzensky" },
  { jmeno: "Praha", kraj: "Středočeský" },
  { jmeno: "Plzeň", kraj: "Plzeňský" },
  { jmeno: "Brno", kraj: "Jihomoravský" }
];
const [{ kraj }] = Mesta;
console.log(kraj); // Plzeňský
```

Zdroj: Vlastní kód

Závěr

Řetězce objektů jsou často používanou datovou strukturou v programování. Tento příklad slouží k pochopení toho, jak můžeme jednoduše získat data z těchto řetězců. Na první pohled se zápis „[{kraj}]" může zdát komplikovaný. Je důležité si uvědomit, jak kombinace těchto závorek funguje v praxi. Pojďme se na to tedy podívat podrobněji. Pokud při destructuring použijeme hranaté závorky, tak dostaneme prvek v poli viz kód níže.

```
const [ kraj ] = Mesta;
console.log(kraj);
```

Výpis z konzole

```
Object {
  jmeno: "Tachov",
  kraj: "Plzensky"
}
```

Při použití složených závorek získáme property objektu s jménem, které jsme použili, takže pokud používáme závorky obě, tak nejdříve získáme první object v poli a poté hodnotu property kraj.

3.4.4 POKROČILEJŠÍ DESTRUCTURING

Požadovné dovednosti

- Znalost syntaxe object literals

Cíl

Dále rozvinout znalosti o destructuring.

Zadání

Pomocí destructuring získejte hodnotu první značky automobilu(„Škoda“) z pole.

Řešení

```
const Auta = {
  znacky: ["Škoda", "Hyundai", "Dacia", "Mercedes"]
};
const { znacky: [znacky]} = Auta;
console.log(znacky); // Škoda
```

Zdroj: Vlastní kód

3.5 CLASSES

3.5.1 JEDNODUCHÁ TŘÍDA

Úvod

V tomto příkladu se podíváme na jednoduchou syntax, jak vytvořit třídu.

Požadované dovednosti

- znalost tříd, objektů
- znalost template literals

Cíl

Pochopit fungování tříd a naučit se syntax.

Zadání

V sekci s kódem můžete vidět vytváření instance třídy „Zaci“ a následné zavolání metody uvnitř třídy s touto instancí. Vytvořte třídu s konstruktorem, který bude očekávat objekt s parametry, které vidíte u vytváření nové instance. Poté vytvořte metodu uvnitř třídy, která bude vracet řetězec s údaji takto:
„Žák Dominik Novosad starý 24 let má rád programování“

Kód

```
  const zak = new Zaci({
    jmeno: "Dominik",
    prijmeni: "Novosad",
    vek: "24",
    oblibenyPredmet: "Programovani",
  });
  console.log(zak.vypisStudenta());
```

Řešení

```
class Zaci {
    constructor(objekt) {
      this.jmeno = objekt.jmeno;
      this.prijmeni = objekt.prijmeni;
      this.vek = objekt.vek;
      this.oblibenyPredmet = objekt.oblibenyPredmet;
```

```
    }
    vypisStudenta() {
      return `Žák ${this.jmeno} ${this.prijmeni} starý ${this.vek} let má rá
d ${this.oblibenyPredmet}  `;
    }
  }
  const zak = new Zaci({
    jmeno: "Dominik",
    prijmeni: "Novosad",
    vek: "24",
    oblibenyPredmet: "Programovani",
  });
  console.log(zak.vypisStudenta());
```

Zdroj kódu: vlastní

Závěr

V tomto příkladu jsme se podívali na vytvoření jednoduché třídy. Zde je důležité si dávat pozor na deklaraci funkce. Pro funkce, které se nachází ve třídě se nepoužívá klíčové slovo function. O kolik je tento zápis jednodušší oproti ES5 se podíváme v následujícím příkladu.

3.5.2 TŘÍDY V POROVNÁNÍ S ES5

Úvod

V předchozím příkladu jsem se podívali na to, jak vytvořit jednoduchou třídu. Nyní se podíváme na to, jak přepsat kód ze starší verze za pomocí tříd a jak moc nám vlastně třídy mohou ulehčit práci.

Požadované dovednosti

- znalost tříd
- povědomí o objektově orientovaném programování(vědět, co znamená super() apod.)

Cíl

- Naučit se přepsat kód ze starších verzí a prohloubit své znalosti o třídách.
- Naučit se určité principy objektově orientovaného programování

Zadání

V sekci s kódem můžete vidět poněkud rozsáhlejší příklad v ES5. Zjednodušte tento příklad za pomocí deklarace tříd z ES6.

Nápověda

Vytvořte si 2 třídy Shape a Circle. Circle je třída, která rozšiřuje předchozí třídu Shape, proto použijte klíčové slovo extends takto:

```
  class Circle extends Shape {
      // kód
}
```

Zavolat konstruktor rodiče můžete za pomocí metody super().

Kód

```
function Shape(id, x, y) {
    this.id = id;
    this.setLocation(x, y);
  }
  Shape.prototype.setLocation = function (x, y) {
    this.x = x;
    this.y = y;
  };
  Shape.prototype.getLocation = function () {
    return {
      x: this.x,
      y: this.y,
    };
  };
  Shape.prototype.toString = function () {
    return 'Shape("' + this.id + '")';
  };
  function Circle(id, x, y, radius) {
    Shape.call(this, id, x, y);
    this.radius = radius;
  }
  Circle.prototype = Object.create(Shape.prototype);
  Circle.prototype.constructor = Circle;
  Circle.prototype.toString = function () {
    return "Circle > " + Shape.prototype.toString.call(this);
  };
  var myCircle = new Circle("mycircleid", 100, 200, 50); // create new insta
nce
  console.log(myCircle.toString()); // Circle > Shape("mycircleid")
```

```
  console.log(myCircle.getLocation()); // { x: 100, y: 200 }
```

Zdroj kódu: převzatý z

https://gist.github.com/remarkablemark/fa62af0a2c57f5ef54226cae2258b38d

Řešení

```
class Shape {
    constructor(id, x, y) {
      this.id = id;
      this.setLocation(x, y);
    }
    setLocation(x, y) {
      this.x = x;
      this.y = y;
    }
    getLocation() {
      return {
        x: this.x,
        y: this.y,
      };
    }
    toString() {
      return `Shape("${this.id}")`;
    }
  }
  class Circle extends Shape {
    constructor(id, x, y, radius) {
      super(id, x, y); // call Shape's constructor via super
      this.radius = radius;
    }
    toString() {
      return `Circle > ${super.toString()}`;
    }
  }
  var myCircle = new Circle("mycircleid", 100, 200, 50); // create new insta
nce
  console.log(myCircle.toString()); // Circle > Shape("mycircleid")
  console.log(myCircle.getLocation()); // { x: 100, y: 200 }
```

Zdroj kódu: převzatý z

https://gist.github.com/remarkablemark/fa62af0a2c57f5ef54226cae2258b38d

Závěr

V tomto příkladu můžeme vidět, jak obrovský přínos vlastně deklarace tříd pro nás má. Pro porovnání rozdíl znaků bez mezer je u ES6 o 195 menší.

3.6 MODULES

Úvod

V tomto příkladu zkusíme použít kód ze sekce o destructuringu jakožto modul z jiného souboru.

Požadované dovednosti

- Znalost syntaxe modulů

Cíl

Naučit se používat moduly

Zadání

Níže můžete vidět soubor ve kterém se nachází funkce pro vypočítání důchodu. Vytvořte nový soubor, kde za pomocí modulů naimportujete tuto funkci a vypíšete její výsledek na konzoli.

Řešení

Soubor s funkcí

```
export function vypocitejRokDuchodu(rok) {
  const vek = new Date().getFullYear() - rok;
  return [vek, 65 - vek];
}
```

Zdroj kódu: vlastní

Soubor, kde naimportujeme modul

```
import { vypocitejRokDuchodu } from "module";

const [vek2, duchod] = vypocitejRokDuchodu(1996);
console.log("Věk: ", vek2);
console.log("Důchod: ", duchod);
```

Zdro kódu: vlastní

Závěr

Moduly nám umožňují výrazně zpřehlednit a strukturovat náš kód. Exportovat můžeme funkce, třídy, ale třeba i konstanty. Pokud nejsou soubory ve stejné složce, je třeba uvést cestu k souboru namísto pouhého jména. Pokud chceme používat relativní cestu k souboru je třeba použít „./“

```
import  { vypocitejRokDuchodu  } from "./src/module";
```

Při zápisu absolutní cesty stačí použít jen „/“.

Pokud chceme importovat třídu, stačí pouze za slovem import přidat její jméno takto:

```
import User from "module";
```

Tato syntax lze i kombinovat dohromady s funkcemi přidáním čárky

```
import User, { vypocitejRokDuchodu } from "module";
```

Dále pokud chceme, aby měla naimportovaná funkce jiné jméno, tak stačí přídat klíčové slovo „as“ a jméno, které požadujeme takto:

```
import User, { vypocitejRokDuchodu as  countRetirement } from "module";
```

3.7 ARROW FUNKCE

3.7.1 ARROW FUNKCE S JEDNÍM PARAMETREM A VÝRAZEM

Úvod

Arrow funkce nám umožňují výrazně zkrátit délku kódu u funkcí. Syntax pro zápis arrow funkce se může lišit podle počtu parametrů a výrazů. Funkce pouze s jedním výrazem a jedním parametrem má syntax nejjednodušší a proto začneme právě s ní.

Požadované dovednosti

- znalost arrow funkcí

Cíl

Naučit se základní syntax arrow funkce

Zadání

Přepište níže nalezenou funkci pro vynásobení proměnné 5 arrow funkcí. Poté tuto funkci zavolejte a výsledek vypište do konzole.

Kód

```
const Nasobeni = function(cislo){
    return cislo * 5;
    }
```

Zdroj: Vlastní kód

Řešení

```
const Nasobeni = cislo => cislo * 5;

const Nasobek = Nasobeni(6);
console.log("Výsledek je: ",Nasobek); //30
```

Zdroj: Vlastní kód

Závěr

Pokud má arrow funkce pouze jeden parametr, tak není třeba tento parametr uvádět v závorkách. Pokud by se jednalo o funkci, která dostane 2 čísla ,která vynásobí, tak by bylo třeba tyto parametry umístit do závorek a oddělit je čárkou takto:

```
const Nasobeni = (cislo1,cislo2) => cislo1 * cislo2;
```

```
const Nasobek = Nasobeni(6,5);
console.log("Výsledek je: ",Nasobek); //30
```
Zdroj: Vlastní kód

Jelikož má funkce jenom jeden výraz, tak není třeba tělo funkce vkládat do složených závorek. Takto psaná funkce nám bude vracet výsledek výrazu, který se nachází za šipkou i bez klíčového slova return.

Pokud bychom psali funkci s více výrazy, např. kromě dělení bychom chtěli vypsat něco do konzole, tak už je třeba do funkce přidat složené závorky a tedy i return takto:

```
const Nasobeni = cislo =>{

   let vysledek = cislo1 * 5;
   console.log("Příklad úspěšně vypočítán");
   return vysledek
}
```
Zdroj: Vlastní kód

3.7.2 SPRÁVNÉ POUŽITÍ ARROW FUNKCÍ

Úvod

V předchozím příkladu jsme si ukázali, jak nám mohou arrow funkce výrazně zjednodušit zápis našeho kódu, nicméně to není vše, co tyto funkce nabízí. V tomto příkladu se podíváme na další možné využití arrow funkcí.

Požadované dovednosti

- znalost arrow funkcí
- schopnost pracovat s objekty
- znalost funkce map

Cíl

Naučit se správně používat arrow funkce

Zadání

Níže naleznete kód s objektem třídy. V tomto objektu se nachází funkce, která by měla postupně vypsat jednotlivé členy třídy do konzole. Nicméně při volání této funkce

dostáváme chybovou zprávu „Typerror: cannot read property „NĚCO“ of undefined“. Využijte vašich předchozích dovedností o arrow funkcích k vyřešení tohoto problému.

Kód

```
const trida = {
    jmenoTridy: "9.C",
    zaci: ["Pavel", "Jirka", "Jakub", "Tereza"],
    vypisTridy: function(){
      return this.zaci.map(function(zak) {
        return `${zak} je ve tride ${this.jmenoTridy}`;
      });
    }
  };

  console.log(trida.vypisTridy()); // Typerror: cannot return value jmenoTri
dy of undefined
```

Zdroj: Vlastní kód

Řešení

```
const trida = {
    jmenoTridy: "9.C",
    zaci: ["Pavel", "Jirka", "Jakub", "Tereza"],
    vypisTridy: function(){
      return this.zaci.map((zak) => {
        return `${zak} je ve tride ${this.jmenoTridy}`;
      });
    }
  };

  console.log(trida.vypisTridy); //"Pavel je ve tride 9.C", "Jirka je ve tri
de 9.C", "Jakub je ve tride 9.C", "Tereza je ve tride 9.C"
```

Zdroj: Vlastní kód

Závěr

Jak už určitě dávno víte, pokud pracujete v kódu s funkcí, ve které používáte hodnotu nějaké proměnné, tak se kód podívá ve scopu funkce po deklaraci této proměnné a pokud jí najde, tak jí použije. Pokud jí nenajde, tak se po ní podívá ve scopu vně funkce a tak pokračuje, dokud nedojde ke globálnímu scopu. Klasické funkce definují hodnotu this za nás implicitně, proto ji nikdy nebudou hledat mimo funkci, jelikož je definována uvnitř. Arrow funkce za nás hodnotu this nedefinují a proto ji defaultně budou hledat ve scopu

okolo funkce jako u jakékoliv jiné proměnné, a proto je arrow funkce v tomto případě mnohem vhodnějším a jednodušším řešením.

Pokud bychom chtěli tento problém vyřešit bez pomocí arrow funkcí, tak je tu možnost uložit hodnotu this do proměnné uvnitř funkce na které se potom budeme odkazovat při volání hodnoty jmenoTridy.

```
const trida = {
    jmenoTridy: "9.C",
    zaci: ["Pavel", "Jirka", "Jakub", "Tereza"],
    vypisTridy: function(){
       let _this = this;
      return this.zaci.map(function(zak) {
        return `${zak} je ve tride ${_this.jmenoTridy}`;
      });
    }
  };

  console.log(trida.vypisTridy()); //"Pavel je ve tride 9.C", "Jirka je ve t
ride 9.C", "Jakub je ve tride 9.C", "Tereza je ve tride 9.C"
```

Zdroj: Vlastní kód

3.7.3 KDY NEPOUŽÍVAT ARROW FUNKCE

Požadované dovednosti

- Znalost arrow funkcí, tříd, konstruktorů

Cíl

Naučit se, kdy je vhodné zvolit arrow funkce

Zadání

Přepište následující kód pomocí arrow funkcí tam, kde to dává smysl.

Kód

```
var Entity = function( name, delay) {
    this.name = name;
    this.delay = delay;

};

Entity.prototype.greet = function(){
    setTimeout(function() {
```

```
    console.log("Hi I am: ",this.name);

    }.bind(this), this.delay );

};

var java = new Entity("Java", 5000);
var cpp = new Entity("C++", 30);


java.greet();
cpp.greet();
```

Zdroj: Kód převzatý z https://www.youtube.com/watch?v=rmNRLGgUiAo

Řešení

```
var Entity = function( name, delay) {
    this.name = name;
    this.delay = delay;

};

Entity.prototype.greet = function(){
    setTimeout( () => {
    console.log("Hi I am: ",this.name);

    }, this.delay );

};

var java = new Entity("Java", 5000);
var cpp = new Entity("C++", 30);


java.greet();
cpp.greet();
```

Zdroj: Kód převzatý z https://www.youtube.com/watch?v=rmNRLGgUiAo

Závěr

V sekci s tímto kódem máme 3 funkce. U první funkce Entity není vhodné použít arrow funkci, protože se jedná o konstruktor a potřebujeme daný kontext pro vytváření objektů. Stejné pravidlo platí i pro funkci greet. Arrow funkci ale můžeme využít u funkce setTimeout, kde můžeme poté odstranit „.bind(this)“, protože to dělá arrow funkce za nás.

3.8 PROMISES

3.8.1 JEDNODUCHÝ PROMISE

Úvod

Promises nám umožňují psát asynchronní kód mnohem zřetelněji a spolehlivěji než bylo možné v předchozích verzích JavaScriptu, kde se tyto kódy psaly většinou za pomocí callbacků. Četné použití callbacků často kód výrazně komplikovalo a bylo mnohem těžší takový kód upravovat, natož se v něm vyznat. V tomto příkladu si zatím vyzkoušíme vytvořit jednoduchý promise.

Požadované dovednosti

- Znalost promises

Cíl

Naučit se základní syntax promises

Zadání

Vytvořte jednoduchý promise, který se splní za předpokladu, že násobek 2 vámi zvolených proměnných bude větší nebo rovno 20. Poté vypište do konzole zprávu podle toho, jestli byl promise resolved nebo rejected.

Řešení

```
let p = new Promise((resolve, reject) => {
    let a = 5;
    let b = 4;
    if (a * b >= 20) {
      resolve("Sucess");
    } else {
      reject("Error");
    }
  });
  p.then((message) => {
    console.log("Vysledek promisu: ", message);
  }).catch((message) => {
    console.log("Vysledek promisu: ", message);
  });
```

Zdroj: Vlastní kód

3.8.2 NÁHRADA ZA CALLBACKY

Požadované dovednosti

- Znalost promises

Cíl

Naučit se přepsat kód ze starších verzí JS a rozšířit dosavadní dovednosti o promises

Zadání

Níže můžete vidět kód, který využívá callbacků pro výpis úspěšné či chybové zprávy na základě deklarovaných konstant. Zjednodušte tento kód s pomocí promises.

Kód

```
const produktNacten = true;
const klientNacten = true;
function dataNactenaCallback(callback, errorCallback) {
  if (!produktNacten) {
    errorCallback({
      jmeno: "Produkt není načten.",
      zprava: "Před prací musí načíst objekt s produktem.",
    });
  } else if (!klientNacten) {
    errorCallback({
      jmeno: "Klient není načten.",
      zprava: "Před prací musí načíst objekt s klientem.",
    });
  } else {
    callback("Vše je načteno. Můžete pracovat");
  }
}
dataNactenaCallback(
  (zprava) => {
    console.log("Úspěch: ", zprava);
  },
  (chyba) => {
    console.log(chyba.jmeno, chyba.zprava);
  }
);
```

Zdroj kódu: Převzatý z https://www.youtube.com/watch?v=DHvZLI7Db8E

Řešení

```
function dataNactenaPromise() {
  return new Promise((resolve, reject) => {
    if (!produktNacten) {
      reject({
        jmeno: "Produkt není načten.",
        zprava: "Před prací musí načíst objekt s produktem.",
      });
    } else if (!klientNacten) {
      reject({
        jmeno: "Klient není načten.",
        zprava: "Před prací musí načíst objekt s klientem.",
      });
    } else {
      resolve("Vše je načteno. Můžete pracovat");
    }
  });
}
dataNactenaPromise()
  .then((zprava) => {
    console.log("Úspěch: ", zprava);
  })
  .catch((chyba) => {
    console.log(chyba.jmeno, chyba.zprava);
  });
```

Zdroj: kód převzatý z https://www.youtube.com/watch?v=DHvZLI7Db8E

Závěr

Jak můžete vidět, řešení s pomocí promises je velmi podobné. Stačí jen nahradit funkce callback a errorCallback za resolve a reject a poté je správně zachytit pomocí .then a .catch.

3.8.3 PROMISE .ALL

Úvod

Představte si, že pracujete na složitější webové aplikaci, která má v sobě naimplementovanou databázi s klienty, vašim produktem a objednávkami a snažíte se vytvořit pro zaměstnance stránku, kde budou mít viditelná všechna dostupná data. Než

budeme chtít tato data jakkoliv zobrazovat, musíme se ujistit, že jsou veškerá tato data načtena. Načítání každé z jednotlivých skupin dat(klienti, produkt, objendávky) můžeme vyřešit přes jednotlivě přes promises, kde budeme mít pro každou z těchto skupin jeden promise. Jak ale potom s výsledky těchto promise naložit? Co když jeden z nich selže a tudíž bude rejected, zatímco ostatní se úspěšně podaří splnit? Promise api má na tento problém přesně udělanou funkci s názvem Promise.all s kterou jsme se už seznámili v teoretické části a právě na tuto funkci se v tomto příkladu podíváme.

Požadované dovednosti

- Znalost promises, promise.all

Cíl

Naučit se pracovat s funkcí Promise.all()

Zadání

Vytvořte tři jednoduché promises, které poté použijete jako parametr funkce promise.all, jejíchž výsledek poté vypíšete do konzole.

Řešení

```
let promise1 = () => Promise.resolve("Seznam uživatelů načten.");
let promise2 = () => Promise.resolve("Seznam objednávek načten.");
let promise3 = () => Promise.resolve("Seznam produktů načten.");

Promise.all([promise1(),promise2(),promise3()]).then((vyslednePole) => {
      for(let i = 0; i < vyslednePole.length;i++){
       console.log(vyslednePole[i]);
      }
})
```

Zdroj: Vlastní kód

Závěr

Promise.all je resolved pouze pokud jsou všechny promises,které jsme zvolili jako parametr také úspěšně resolved. Pokud by jakýkoliv z nich byl rejected, byl by rejected i promise.all. Z tohoto důvodu se jedná o skvělou pomůcku v tomto případě, kdy potřebujeme mít načtená veškerá data než budeme postupovat dále.

3.9 REKAPITULACE ZÍSKANÝCH DOVEDNOSTÍ

3.9.1 KOMPLEXNÍ PŘÍKLAD

Úvod

V tomto příkladě si vyzkoušíme pracovat hned s několika možnostmi představenými v předchozích kapitolách.

Požadované dovednosti

- promises, promise.all()
- let a const
- destrukturování
- moduly
- třídy
- template literals
- syntax arrow funkce

Cíl

Zopakovat si získané dovednosti.

Zadání

Pro práci na tomto příkladu budete potřebovat dva JavaScriptové soubory. V prvním souboru se inspirujte příkladem ze sekce o classes, kde bylo za úkol vytvořit jednoduchou třídu. Vytvořte podobnou třídu, jenom tentokrát využijte destrukturování pro získání hodnot objektu. Po vytvoření třídu vyexportujte.

V druhém souboru dokončenou třídu naimportujte a vytvořte 2 instance této třídy s libovolnými property například takto:

```
const zak = new Zaci({
    jmeno: "Dominik",
    prijmeni: "Novosad",
    vek: "24",
    oblibenyPredmet: "Programovani"
  });
```

Dále vytvořte 2 promises. V každém z nich zavolejte přes jednu z vytvořených instancí metodu "vypisStudenta()" s tím, že promise bude resolved pokud zavolání metody proběhlo bez problému. Hodnota, kterou promise bude vracet při úspěšném dokončení je návratovou hodnotou z metody "vypisStudenta()". Nakonec vytvořte Promise.all(), který bude mít jako parametry předchozí 2 promises. Pokud bude Promise.all() úspěšně resolved, vypište výsledek do konzole.

Řešení

```
Modul.js

export class Zaci {
    constructor(objekt) {
      this.jmeno = objekt.jmeno;
      this.prijmeni = objekt.prijmeni;
      this.vek = objekt.vek;
      this.oblibenyPredmet = objekt.oblibenyPredmet;
    }
    vypisStudenta() {
      return `Žák ${this.jmeno} ${this.prijmeni} starý ${this.vek} let má rá
d ${this.oblibenyPredmet}  `;
    }
  }
```

```
Main.js

import Zaci from Modul.js;

  const zak = new Zaci({
    jmeno: "Dominik",
    prijmeni: "Novosad",
    vek: "24",
    oblibenyPredmet: "Programovani"
  });

  const zak2 = new Zaci({
    jmeno: "Jan",
    prijmeni: "Novák",
    vek: "22",
    oblibenyPredmet: "DevOps"
  });

  let promise1 = new Promise((resolve, reject) => {
    let a = zak.vypisStudenta();
    if (a) {
```

```
    resolve(a);
  } else {
    reject("Error v promise1");
  }
});

let promise2 = new Promise((resolve, reject) => {
  let a = zak2.vypisStudenta();
  if (a) {
    resolve(a);
  } else {
    reject("Error");
  }
});

Promise.all([promise1, promise2])
  .then((vyslednePole) => {
    for (let i = 0; i < vyslednePole.length; i++) {
      console.log(vyslednePole[i]);
    }
  })
  .catch((vyslednePole) => {
    console.log("Promise selhal: ", vyslednePole);
  });
```

Závěr

Tento příklad slouží k finálnímu zopakování většiny probraných možností. Přestože vypadá příklad na první pohled velmi komplexní, pokud jste nenarazili na problém u předchozích příkladů, tak by vám ani tento neměl dělat problém, protože nepředstavuje nic, co by doposud nebylo vyřešeno.

ZÁVĚR

Cílem této bakalářské práce bylo představit nové možnosti v ECMAScript 6 pro JavaScript. K tomu abych tento cíl mohl řádně splnit, bylo nejdříve třeba se v první kapitole seznámit s historií JavaScriptu, kde jsem popsal celou cestu, kterou JavaScript prošel od svého vytvoření Brendan Eichem až po verzi ES6.

V druhé kapitole se věnuji jednotlivým novým možnostem, které vždy jednotlivě představím a uvedu, co nám vlastně přináší a proč je do JavaScriptu vůbec přidali. U každé této sekce také názorně ukážu, jaká je správná syntax pro tento prvek a jak se používá. Jsou zde znázorněny možnosti, které nám nabízí pouze kratší syntax (syntactic sugar), i takové, které nabízí zcela nové funkcionality.

V praktické části jsem vytvořil sadu příkladů, které slouží pro procvičení a pochopení daných možností. Příklady se soustředí na znázornění rozdílů od předchozích verzí a specifické užití pro nové možnosti, která nám tato verze přináší. U každé z nových možností je alespoň jeden příklad, ve kterém se krátce procvičí základní použití a syntax. U dalších příkladů už jde spíše o konkrétnější užití, kde ukážu různé případy, kdy je vhodné či nevhodné tyto nové funkce využít, nebo ukážu další dodatečné funkcionality, které nabízí.

Budoucnost JavaScriptu je rozhodně velmi slibná. V současnosti pracuje více než většina front-endových vývojářů na JavaScriptu. Velké oblibě se také v současnosti těší frameworkům jako Vue, Angular a React, které nám dále usnadňují práci s JavaScriptem. Dále díky frameworku Node.js jsme schopní využívat JavaScript i ze strany serveru. Když vezmeme všechny tyto faktory v potaz spolu s trendy souvisejícími s průmyslem 4.0, můžeme nepochybně očekávat, že Javascript z žebříčku nejpoužívanějších programovacích jazyků nikam nepůjde a můžeme tedy v budoucnu očekávat vydání dalších rozšíření, či vývoj nových frameworků.

SEZNAM LITERATURY

HARRISON, Ross. *ECMAScript Cookbook* [online]. 2018. [cit. 2020-01-19]. ISBN 9781788628174.

ZAKAS, Nicholas C. Understanding ECMAScript 6: the definitive guide for JavaScript developers. San Francisco: No Starch Press, [2016]. ISBN 978-1-59327-757-4.

S. ENGELSCHALL, Ralf. *ECMAScript 6 — New Features: Overview & Comparison* [online]. 2015. Dostupné z: http://es6-features.org/#Constants

GROVER, Deepak a Hanu PRATEEK KUNDURU. ES6 for humans: the latest standard of JavaScript: ES2015 and beyond. Berkeley, California?: Apress, [2017]. Books for professionals by professionals. ISBN 978-1-4842-2622-3.

BEVACQUA, Nicolas. Practical modern Javascript. Sebastopol, CA: O'Reilly Media, [2017]. ISBN 978-1491943533.

BEVACQUA, Nicolas. *Modular JS: Practical ES6* [online]. O'Reilly Media, 2016 [cit. 2020-04-29]. ISBN 9781491943472.

ES6 - Overview [online]. 2018. Dostupné z: https://www.tutorialspoint.com/es6/es6_overview.htm

OGDEN, Max. *Callback Hell* [online]. 2012. Dostupné z: http://callbackhell.com/

REMARKABLE, Mark. *JavaScript Classes - ES5 vs ES6* [online]. New York, 2019. Dostupné z: https://gist.github.com/remarkablemark/fa62af0a2c57f5ef54226cae2258b38d

RAUSCHMAYER, Axel. *Exploring ES6: Upgrade to the next version of JavaScript*. 2015.

PRUSTY a NARAYAN. *Learning Ecmascript 6*. Packt Publishing, 2015. ISBN 9781785884443.

HARRISON, Matt. Tiny ES6 Notebook: Curated JavaScript Examples. 2017. ISBN 9781973738589.

PŘÍLOHY

https://codepen.io/ECMAScript6