

JavaScript: řešené úlohy

RNDr. Ondřej Žára

Obsah

Kapitola 1: Úvod	4
Jak pracovat s knihou	4
Pět minut dějepisu	5
Co by měl čtenář vědět	7
Středníky v JavaScriptu	8
Ladění	9
Další čtení	12
Materiály online	13
Kapitola 2: První setkání	14
Úloha	14
Zelenáči: drobná vylepšení	18
Koumáci: průzkum stromu DOMu	20
Profíci: druhy skriptů	25
Kapitola 3: Pole a iterace	29
Úloha	29
Zelenáči: anonymní a arrow funkce	31
Koumáci: druhy iterací	33
Profíci: scope a closure	37
Kapitola 4: Kontrola formuláře	41
Úloha	41
Zelenáči: další druhy událostí	46
Koumáci: výjimky	50
Profíci: zpožděná kontrola při psaní	55
Kapitola 5: HTTP na pozadí	59
Úloha	59
Zelenáči: řízení toku kódu pomocí operátorů	65
Koumáci: <i>falsy values</i> a operátory s implicitním přetypováním	67
Profíci: <i>same-origin policy</i> a CORS	70
Kapitola 6: Cesta k SPA, riziko XSS	74
Úloha	74
Zelenáči: práce s adresním řádkem	79
Koumáci: funkcionální iterace	83
Profíci: <i>fetch</i> , <i>Promises</i> a <i>async/await</i>	86

Kapitola 7: SPA administrační systém	92
Úloha	92
Zelenáči: event delegation	95
Koumáči: bublání a zachytávání	99
Profíci: o řetězcích a číslech	103
Kapitola 8: Třídy a objekty	106
Úloha	106
Zelenáči: odebírání posluchačů událostí	111
Koumáči: klíčové slovo this	116
Profíci: prototypová dědičnost	120
Kapitola 9: Web Components	128
Úloha	128
Zelenáči: gettery a settery	132
Koumáči: předávání hodnotou a odkazem	134
Profíci: vlastní události	136
Kapitola 10: Intl, Storage, polyfilly a další API	141
Úloha	141
Zelenáči: Web Storage	146
Koumáči: Polyfilly	148
Profíci: další API	152
Kapitola 11: HTML Canvas	153
Úloha	153
Zelenáči: další dovednosti canvasu	160
Koumáči: obrazová data z kamery a videa	163
Profíci: kdy s canvasem narazíme a jak z toho ven	165
Kapitola 12: JavaScript mimo prohlížeč	173
Úloha	173
Zelenáči: npm	179
Koumáči: tooling	182
Profíci: alternativy k Node.js	186
Kapitola 13: Když JavaScript nestačí	188
Zelenáči: React	188
Koumáči: TypeScript	194
Profíci: WebAssembly	199
Kapitola 14: Co se nevešlo	202
Zelenáči: SVG	202
Koumáči: striktní režim	206
Profíci: iterační protokol	208

Úvod

Jen těžko se lze pohybovat v oblasti tvorby webu a nepovšimnout si, jak značnému zájmu se JavaScript těší. Spolu s oblíbeností a rozšířeností logicky roste také poptávka po kvalifikovaných vývojářích. Najít zkušeného JavaScriptového programátora přitom vůbec není jednoduché; základy si na Internetu najde každý, ale skutečně hlubokého pochopení dojde jen málokdo. Je k tomu zapotřebí mnoho praxe, zvědavosti i obecného algoritmického rozhledu.

Na oblíbeném programátorském webu stackoverflow.com je položeno přes dva a půl milionu otázek na téma JavaScript. To je dobrým dokladem zájmu i nedostatečného vzdělání mezi webovými vývojáři. Ambicí této knihy je poskytnout podklady pro systematické pochopení JavaScriptu a jeho praktického používání.

Konkrétní programátorské nástroje, postupy a knihovny přicházejí a s různorodým odstupem zase mizí. Základní koncepty a jazyky WWW však zůstávají a postupným vylepšováním upevňují web na předním místě každodenních životních činností. Práce s HTML stránkami, jejich design a postupné ožívování pomocí skriptů je tak více a více považováno za základní IT vzdělání; mnoho vývojářů se s prvními algoritmy a jazyky potká právě v rámci tvorby webu.

S rostoucí nabídkou webových technologií je na místě dodávat i užitečné studijní podklady. Zde přichází ke slovu tato kniha: poskytuje krátký exkurz do světa pod pokličkou HTML stránek, ukazuje principy vývoje a seznamuje s typickými postupy. Na své by si tak měli přijít jak začátečníci, tak i zkušenější vývojáři z ostatních oborů, kteří by rádi obhlédli, jak že se to pracuje s tím HTML a především JavaScriptem.

Jak pracovat s knihou

S výjimkou této jsou všechny kapitoly v knize postaveny stejným způsobem: každá se točí okolo jedné konkrétní úlohy, která je zmíněna vždy v úvodu kapitoly. Následuje vzorové řešení a jeho krátký rozbor. Úloha je volena tak, aby

připomínala opravdové zadání, se kterým se můžeme v praxi setkat. Zároveň s postupem času roste obtížnost úloh, abychom se mohli seznamovat s dalšími vlastnostmi jazyka.

V každé kapitole pak následuje trojice podkapitol, které využívají řešenou úlohu jako místo pro vysvětlení dalších konceptů JavaScriptu. Tyto podkapitoly jsou seřazeny dle obtížnosti a obdobným způsobem pojmenovány:

- Každá první podkapitola je určena pro **zelenáče**. Pozor, nenechme se takovým označením zmást, ani urazit! Jde prostě o ty, kteří s JavaScriptem začínají a teprve se rozkoukávají. Podkapitoly pro zelenáče představují základní vlastnosti JavaScriptu a zpravidla se v nich naučíme různé formy vylepšení vzorového řešení úlohy.
- Každá druhá podkapitola je určena pro **koumáky**. Je v ní prostor pro obecnější vysvětlení probíraných témat. Namísto použití JavaScriptu (jen) pro splnění konkrétního zadání se koumáci seznámí i s alternativami a pokročilejšími koncepty. Tyto podkapitoly pomůžou rozšířit obzory těm, kterým přijde vzorové řešení přímočaré a chtěli by vědět více.
- Každá třetí podkapitola je určena pro **profíky**. Probíráme v ní ty rysy jazyka, které jsou buď komplikované, nebo se s nimi nesetkáváme moc často. Profíci se v těchto podkapitolách mohou seznámit s tématy, o kterých mají frontendoví vývojáři často jen neúplné znalosti, případně se jich intuitivně straní a buď je ignorují, nebo je obchází používáním knihovního kódu.

Cílem knihy je nabídnout užitečný obsah všem zájemcům o JavaScript. Projděte si prvních pár kapitol a *nakalibrujte se* na jednoho ze tří vzorových čtenářů – podle toho pak můžete dalším podkapitolám věnovat větší pozornost, nebo je jen v rychlosti prolétnout a přeskočit.

Pět minut dějepisu

Historie programovacího jazyka možná nepatří mezi nejzábavnější části knihy, ale může přinést užitečný kontext k informacím, které se dozvíme v dalších kapitolách. JavaScript navíc v průběhu života dostával různé názvy a zkratky, se kterými se v rámci tohoto dějepisného rychlokurzu seznámíme.

Archeologičtí nadšenci se mohou detailnější informace dozvědět mimo jiné na stránce <https://en.wikipedia.org/wiki/JavaScript#History> – tato podkapitola je zkrácenou kombinací textů z několika zdrojů, zejména však toho odkazovaného. Pojďme si nyní představit klíčové body v historii JavaScriptu.

- První verze jazyka JavaScript byla navržena a naimplementována během pouhých deseti dní (!) Brendanem Eichem ve společnosti Netscape. Psal se rok 1995 (mimo jiné též rok vzniku jazyka PHP) a Eich dostal za úkol v této kruté časové lhůtě navrhnout *menšího a hloupějšího sourozence Javy* – jako konkurenční krok v tvrdém souboji s prohlížečem Internet Explorer. Tato verze jazyka se jmenovala Mocha a záměrně, aby nevypadala příliš jako Java, nabízela namísto tříd prototypy. K jejich fungování Eicha inspiroval jazyk Self; právě prototypová dědičnost se následně stala definující vlastností, kterou se JavaScript dodnes odlišuje od ostatních objektově-orientovaných jazyků. Syntaxi navrhl po vzoru Javy a C; z jazyka Scheme převzal možnosti funkcionálního programování a uzávěry (viz třetí kapitolu). Během integrace jazyka do prohlížeče Netscape Navigator došlo ještě v roce 1995 ke dvěma změnám jména: nejprve na LiveScript, poté už na finální JavaScript.
- V roce 1996 se objevuje první konkurenční implementace – JScript v Internet Exploreru 3. Zároveň dochází k formální standardizaci jazyka, tentokrát pod názvem ECMAScript (ECMA je název standardizační organizace *European Computer Manufacturers Association* zodpovědné mimo jiné za specifikace formátů dat na CD či jazyka C#). Až do dnešního dne tak žijeme v nezvyklém terminologickém guláši: striktně vzato pracujeme s implementací respektující předpisy standardu ECMAScript, ale implementace jako taková (nezávisle na použitém prohlížeči či serverovém prostředí) vykonává zdrojový kód jazyka JavaScript. Tato dualita bude patrně přetrvávat i nadále, mimo jiné proto, že označení JavaScript je chráněnou obchodní značkou firmy Oracle.
- Firma Microsoft v roce 2000 vydává webový e-mailový klient Outlook Web Access; pro jeho potřeby do prohlížeče Internet Explorer 5 přidává nové proprietární rozhraní, které se následně dočká názvu XMLHttpRequest. Tím dochází k dramatickému nárůstu možností JavaScriptu (do té doby omezeného fakticky na kontrolu webových formulářů před jejich odesláním) a následně k rozvoji jeho popularity mezi webovými vývojáři. Začíná tak období boomu tzv. *Ajaxu* (více si o tom povíme v kapitole 5).

- Pro svůj vlastní prohlížeč Chrome se společnost Google rozhodla vytvořit zbrusu novou a výkonnou implementaci JavaScriptu. V roce 2008 se objevuje Chrome 1.0 a implementace dostává název V8. Mimo jiné i díky rychlosti V8 dochází k oživení myšlenky, že jazyk jako takový nemá důvod být používán pouze jako přívěsek webového prohlížeče; není žádná technologická překážka k použití JavaScriptu i pro programování serverového kódu. Hned v roce 2009 se objevuje projekt Node.js (jeho autorem je Ryan Dahl): sada rozhraní a knihoven obalující implementaci V8 a dovolující tak jednoduše spouštět JavaScriptový kód i mimo webový prohlížeč.
- Rok 2015 je pro JavaScript pravděpodobně nejdůležitější milník v moderních dějinách jazyka. Po mnoha letech vývoje *do šuplíku* konečně dochází ke zveřejnění nové (šesté) verze jazyka. V porovnání s minulými verzemi přináší tolik novinek, že jejich popis by vydal na vlastní knihu. Tato verze nese označení **ES6** (ECMAScript 6), ale zároveň se jí podle roku vzniku říká **ES2015**. Od tohoto roku organizace ECMA začíná do jazyka integrovat novinky s pravidelnou roční kadencí. Po pár letech už přestává dvojí číslování dávat smysl (např. skutečnost, že verze ES12 je to samé jako ES2021) a k dnešnímu dni jednotlivé verze odlišujeme prostě jen pomocí čísla roku.
- V roce 2018 se objevuje první vážná konkurence pro Node.js; jedná se o projekt *Deno* založený taktéž Ryanem Dahlem. O obou těchto prostředích pro vykonávání JavaScriptu mimo webový prohlížeč si povíme více v kapitole 12. Třetí člen tohoto spolku, projekt *Bun*, je nyní v roce 2024 žhavá technologická novinka.

Co by měl čtenář vědět

Přestože tato kniha nabízí celou řadu kapitol pro začátečníky, není vhodná pro úplné amatéry v řemesle programátorském. Od čtenářů se očekávají znalosti v tomto základním rozsahu:

- Alespoň minimální zkušenost s libovolným programovacím jazykem. Je docela jedno s jakým; většina jazyků si je podobná jako vejce vejci. Dobré jazyky pro úplné začátečníky jsou Python, Go, Java, Ruby, Lua, nebo třeba VBScript, pomocí kterého lze psát makra v kancelářském software.

- Povědomí o tom, k čemu slouží proměnné a jejich datové typy. Znalost řetězců, čísel a pravdivostních hodnot. Představa toho, že data jsou uložena v operační paměti počítače.
- Podstata hlavních logických operací, které při programování potkáváme: podmínky, cykly, funkce. Nezáleží na jejich syntaxi (zápisu), ale na jejich užitku při sestavování programů.
- Fungování webových stránek, tj. jazyk HTML a znalost mechanismu *World Wide Web*, tj. představa internetem propojené počítačové sítě a role klientů a serverů v ní. V jazyce HTML pro nás není důležité velké množství jednotlivých značek, ale koncept toho, že data přenášená po síti mají tvar odlišný od výsledného vizuálu, který vidíme v prohlížeči.
- Ovládání počítače a softwarového vybavení. Pro práci s JavaScriptem nám ve většině kapitol stačí libovolný moderní (aktualizovaný) webový prohlížeč a textový editor. Samozřejmě je možné použít specializovaná vývojová prostředí (IDE), ale rozhodně to není nutné. Stejně tak nezáleží na použitém operačním systému.

Středníky v JavaScriptu

(Ne)používání středníků na koncích řádků je jednou ze specifických vlastností JavaScriptu. Po nějakou dobu si můžeme vystačit se zjednodušujícím tvrzením, že *středníky na konec řádku psát můžeme, ale nemusíme*. Snadno se pak ale stane, že narazíme na konkrétní případ, kdy tím způsobíme nepříjemnou chybu. Proč?

Náš zdrojový kód je tvořen posloupností jednotlivých příkazů. Tyto příkazy od sebe musíme oddělovat; někdy je takové oddělení poznat z použité syntaxe, jindy jej můžeme vynutit zapsáním středníku. Specialitou JavaScriptu je funkcionální nazývání **ASI** (automatic semicolon insertion). Ta říká, že pokud ve zdrojovém kódu mezi příkazy schází potřebné oddělení, interpret se pokusí tento kód *opravit* automatickým vložením středníku. Fakticky to znamená, že nepoužíváním středníků se sice dopouštíme mnoha syntaktických chyb, ale během vykonávání dojde k jejich automatické nápravě.

Bohužel, pravidla pro ASI jsou složitá a mají řadu výjimek. Spoléháním na ASI se vystavujeme riziku, že takovou výjimku potkáme, středník sami neuvedeme, ale následně jej nevloží ani ASI. Abychom nemuseli s komplikovaným algoritmem bojovat, budou v této knize všude středníky na koncích řádků zapsány explicitně.

Ladění

Tvorba zdrojového kódu a zejména proces *učení se* jsou neodmyslitelně svázány s chybami, kterých se – omylem a někdy i záměrně – dopustíme. V různých programovacích jazycích dochází k chybám různými způsoby, ale jedno mají společné: jako autoři kódu o nich chceme vědět, pokud možno co nejdříve, a potřebujeme jim porozumět (abychom je uměli odstranit a do budoucna se jich vyvarovat). Dlouho a zálučně skrytá chyba dokáže potrápiti i zkušeného vývojáře; nováčka pak třeba od dalšího studia i odradit. Proto je nezbytné vědět, jak se ve světě JavaScriptu o chybách dozvíme a jaké možnosti máme pro jejich vyřešení.

Typicky se při práci s JavaScriptem můžeme setkat se dvěma hlavními kategoriemi problémů:

1. Chyby, které se týkají jazyka jako takového. Do této množiny spadají chyby syntaktické (zápis v rozporu s pravidly jazyka), přístup k neexistujícím vlastnostem a funkcím, nechycené výjimky (povíme si o nich v kapitole 4). Pokud se v našem kódu podobná chyba objeví, zpravidla to znamená, že interpret (prohlížeč) sám od sebe nedokáže ve vykonávání pokračovat. Při jistém úhlu pohledu je to dobrá zpráva, neboť prohlížeč nám o tom dá jasně vědět a my jsme si existencí chyby jisti. Stačí ji pak nalézt a opravit.
2. Chyby, které se týkají naší vlastní algoritmizace. Typicky jsou založeny na špatné úvaze, nezahrnuté okrajové situaci (anglicky *edge case*), chybném výpočtu a podobně. Při těchto chybách se kód vykonává *jakoby nic*, ale produkuje špatné výsledky. Zpravidla pak nezáleží na tom, že pracujeme v JavaScriptu, neboť obdobné chyby bychom se pravděpodobně dopustili i v jiném jazyce. I tak nás ale zajímá, jak nám JavaScript může pomoci v odhalení těchto problémů.

Protože klientský JavaScript je vykonáván webovým prohlížečem, bude i hledání chyb pevně svázáno s tímto programem. Nedílnou součástí každého moderního prohlížeče jsou **Nástroje pro vývojáře** (anglicky *Developer Tools*), jejichž rolí je asistence zoufalým programátorům. Seznámení s těmito nástroji je nezbytné pro vyřešení většiny problémů, na které při práci narazíme.

V roce 2024 jsme svědky jisté konvergence vývojářských nástrojů napříč prohlížeči. Vypadají sice různě, ale jejich funkcionalita je víceméně stejná. Můžeme se k nim dostat buď pomocí různých navigačních nabídek v prohlížeči, nebo stiskem klávesových zkratk. Zdaleka nejrozšířenější je zde klávesa **F12**, která vývojářské nástroje pouští v prohlížečích Firefox, Chrome i Edge (ve Windows a některých variantách Linuxu). U ostatních platforem bude muset čtenář sám ověřit, jak přesně se k nim ve svém oblíbeném prohlížeči dostane. Sedíte teď zrovna vedle počítače? Pokud ano, zkuste si vývojářské nástroje otevřít (a následně skrýt) hned teď! Při čtení knihy se to bude ještě mnohokrát hodit.

Tyto *devtools* nabízí funkcionalitu členěnou do záložek; můžou jich být i desítky. Pro potřeby naší knihy se budou nejčastěji hodit záložky:

- **Konzole** (anglicky *Console*), která je primárním mechanismem pro interakci s JavaScriptovým interpretem. Prohlížeč sem vypisuje všechny informace, které považuje za relevantní pro běh programu. Zejména to jsou chyby první kategorie, které se zobrazují červeně. Jakmile vidíme v konzoli červený řádek, velmi často to znamená naši chybu, která zasluhuje pozornost. U chyb bývá vypsáno i číslo řádku zdrojového kódu, na kterém problém nastal. Důležitou součástí konzole je také vstupní pole, které nám dovoluje v omezené míře zadávat a vykonávat vlastní povely v JavaScriptu. Jedná se o tzv. REPL (Read-Eval-Print-Loop), který můžeme znát například z jazyka Python. Pro rychlé vyzkoušení tím pádem nemusíme zasahovat do zdrojových souborů a ukládat je na disk; stačí potřebný příkaz zadat do konzole. Tento kód však samozřejmě nebude nikam uložen a nestane se součástí programů, které vytváříme.
- **Síť** (anglicky *Network*) zobrazuje veškerou komunikaci po Internetu, kterou prohlížeč provádí. Zahrnuje to jak data přenášená v rámci přesunu mezi stránkami (tzv. *navigaci*), tak i data získávaná prohlížečem bez explicitního vyžádání na pozadí – obrázky, skripty, styly a další. Jakmile začneme pomocí JavaScriptu vytvářet HTTP požadavky (kapitola 5), bude se nám tato záložka velmi hodit.

- **Zdroje** (anglicky *Sources*), případně **Debugger**, nabízí pokročilejší možnosti prozkoumávání právě vykonávaného kódu. V této záložce můžeme pozastavit spuštěný JavaScript, prozkoumávat hodnoty proměnných a podobně.

Při hledání problémů ve zdrojovém kódu je užitečné mít přehled o tom, které funkce a v jakém pořadí se vykonávají, jakých hodnot nabývají proměnné a podobně. Za tímto účelem existují příslušné nástroje, které pomohou programátorovi o zmiňovaných veličinách informovat. Ve světě klientského JavaScriptu je to historicky především funkce **alert** a dále globální objekt **console**.

Volání **alert(promenna)** vypíše v parametru předanou hodnotu do tzv. *modálního* okénka. Jedná se o primitivní mechanismus, kterým můžeme zobrazit potřebná data. Jeho hlavní výhoda je v tom, že pozastaví vykonávání veškerého JavaScriptu (přepne prohlížeč do ne-JavaScriptového *módu*, odtud název *modální*), a my tak máme čas si veličinu prohlédnout bez obavy, že nám nějaká další aktivita uteče. Za zmínku stojí, že vypisované hodnoty jsou automaticky převáděny na řetězec, takže složitější datové struktury nám **alert** nezobrazí. V dřevních dobách JavaScriptu se takové uživatelské rozhraní používalo i za účelem zobrazení informací běžným čtenářům webu, ale dnes by to byla spíš ostuda, takže si funkci **alert** schováme výhradně pro ladící potřeby.

V proměnné **console** se ukrývá řada funkcí, pomocí kterých můžeme z vlastního kódu zapisovat rozmanitá data do konzole, o jejímž působení v rámci devtools jsme se dozvěděli před pár odstavci. Nejčastější a nám nejvíce užitečné jsou **console.log()**, **console.warn()** a **console.error()**. Jejich chování je identické, jen předaná data zapisují do konzole v různých barevných odstínech, aby se tak odlišila závažnost. Tyto funkce jsou *variadické*, tj. můžeme jim předat libovolný počet parametrů. Každý z nich zapíše do konzole a zvládnou libovolně složité datové typy.

Zvědavý čtenář nyní může zkusit objekt **console** prozkoumat (například voláním **console.log(console)**), a najít tak spoustu dalších užitečných funkcí. O všech se následně může dočíst třeba na webu MDN (viz dále).

Další čtení

Tuto knihu jsem napsal, protože mi na českém trhu schází literatura, která by JavaScript vysvětlovala přístupnou formou a zároveň dokázala oslovit zájemce od nováčků až po odborníky. Rozhodně to však neznamená, že kniha pokryje potřeby každého zvědavého čtenáře! Zároveň není v možnostech jednotlivce úplně ovládnout veškeré rysy natolik bohatého jazyka, jakým JavaScript je. Pojdme se podívat na další (anglické) materiály, které mohou přijít vhod.

- Především tato kniha není referenční příručkou, dokumentací ani kompletním strukturovaným popisem gramatiky a vlastností jazyka. Zájemce o takto formální popis můžeme odkázat na oficiální web pracovní skupiny TC39 (<https://tc39.es/>), která vytváří a koordinuje standard, na základě kterého pak vznikají jednotlivé implementace JavaScriptu. Výrazně praktičtější je pak web s historickým označením **MDN** (Mozilla Developer Network) na adrese <https://developer.mozilla.org/en-US/docs/Web/JavaScript> – zde si můžeme přečíst o všech aspektech jazyka, prohlédnout interaktivní ukázky a taktéž k tvorbě dokumentace přispět. Web totiž stojí na otevřeném wiki-systému a je vytvářen dobrovolníky. Velikou výhodou MDN je také propojení s HTML a CSS, neboť tento web se snaží pokrýt úplně všechny aspekty klientského webového vývoje.
- Sada knih **Exploring JS** od Axela Rauschmayera je k dispozici zdarma online na jeho webových stránkách (<https://exploringjs.com/>). Dr. Rauschmayer je velmi zkušený, píše čtivě a pokrývá široké spektrum témat. Jeho knihy se věnují JavaScriptu zevrubně, takže namísto čtení *od začátku do konce* slouží lépe jako referenční příručka.
- Kniha **You Don't Know JS Yet** (autor Kyle Simpson) je populární a čtivá. Existuje v online verzi (<https://github.com/getify/You-Dont-Know-JS>) i tištěná. Věnuje se především záludným partiím JavaScriptu, tj. těm, které bychom v této knize popisovali zejména v podkapitolách pro profíky.
- Poslední zmínku zaslouží kniha **Eloquent JavaScript**, kterou napsal Marijn Haverbeke. Můžeme si ji přečíst na webu <https://eloquentjavascript.net/>, stáhnout v PDF, případně zakoupit vytištěnou. Je velmi obsáhlá a čtenář v ní nalezne značné množství úloh, často výrazně obtížnějších, než v této knize.

Materiály online

Tato kniha je k dispozici v tištěné podobě a také elektronicky. Její digitální verze, společně se všemi podklady nutnými k publikaci, je vystavena na GitHubu. V repozitáři jsou zároveň umístěny i zdrojové kódy, které v následujících kapitolách probíráme a komentujeme. Pro snazší experimenty tak čtenář může navštívit web <https://github.com/ondras/javascript-resene-ulohy>, kde nalezne nejen studijní materiály, ale také *issue tracker* – nástroj na hlášení chyb a nedostatků. Jeho prostřednictvím dokážeme udržovat text elektronické verze knihy i jednotlivých příkladů aktuální a užitečný. Pokud při čtení narazíte na místo, které se zdá podivné, možná jste objevili chybu. Na GitHubu ji můžete nahlásit, nebo třeba zjistit, že již byla nahlášena a v elektronické verzi knihy opravena.

Pro úplnost je na místě dodat, že zveřejnění digitální verze nastává s drobným odstupem několika týdnů po té tištěné. Zdrojové kódy řešených příkladů jsou ale k dispozici ihned.

První setkání

Úloha

Pro fanoušky Karla Gotta chystáme vzpomínkový web, který bude obsahovat i texty písní. Některé jsou ale příliš dlouhé, takže je nutné zobrazit jen prvních několik znaků a zbytek textu skrýt. Po kliknutí na tlačítko se ukáže celý text písně.

Řešení

```
<!-- kapitola-2.html -->
<h1>Mám styl Čendy</h1>
<pre>
Mezi námi je mnoho chvil
A pokusů, abych se ti zavděčil
Jenomže od tebe se člověk moc nedoví
Stále básníš o nějakém svém záhadném Čendovi
</pre>

<script>
let song = document.querySelector("pre");
let text = song.textContent;

song.textContent = text.substring(0, 50) + "...";

let button = document.createElement("button");
button.textContent = "zobrazit celý text";
song.append(button);

function showCompleteText() {
  song.textContent = text;
}
```

```
button.addEventListener("click", showCompleteText);
</script>
```

První setkání s JavaScriptem! Výše uvedený kód představuje řešení naší úlohy, které z textu písňě zobrazí prvních 50 znaků a zbytek skryje. Po kliknutí na tlačítko se odhalí text celý.

Úvodem přiznejme, že se rozhodně nejedná o nejlepší řešení. Zkušený programátor v něm jistě nalezne místa k vylepšení. Některá z nich si ukážeme v první začátečnické podkapitole. Předtím se ale podíváme na celé vzorové řešení řádek po řádku a vysvětlíme, co vše se z něj můžeme naučit.

Krok po kroku

Převážná většina této knihy se týká *klientského JavaScriptu*, tedy toho scénáře, kdy jazyk používáme pro vylepšení webových stránek. Technologicky to znamená, že:

- k vyzkoušení potřebujeme libovolný webový prohlížeč (tzv. *klient*)
- v prohlížeči otevíráme webovou (HTML) stránku, jejíž přímou či nepřímou součástí je JavaScriptový kód
- prohlížeč proto v rámci zobrazení stránky musí také stáhnout a vykonat instrukce obsažené v tomto kódu

Z toho plyne, že není nutné používat žádný specializovaný software, který by JavaScript zpracovával (interpretoval, kompiloval, ...). Naopak je potřeba alespoň trochu rozumět HTML, protože výsledná stránka je s JavaScriptem niterně provázána: kód píšeme proto, aby stránku vylepšoval, ovlivňoval a měnil.

Tím se dostáváme ke skriptu samotnému. Jeho první dva řádky začínají klíčovým slovem **let**, kterým uvozujeme definici proměnných. Vytváříme dvě proměnné (**song** a **text**) a rovnou je naplníme hodnotami pomocí znaku rovnítka (tzv. *operátor přiřazení*). Protože naším cílem je manipulace se stránkou (skrytí části textu), potřebujeme se především dostat k jednotlivým HTML prvkům ve stránce a jejich obsahu. To je ve světě klientského JavaScriptu možné pomocí sady funkcí a objektů souhrnně nazývaných **DOM** – Document Object Model. Jedná se o velice silný nástroj, který budeme využívat v téměř všech kapitolách. Pomocí DOM můžeme do stránky vkládat nový obsah, upravovat existující, nebo klidně stránku celou vymazat a sestavit znova a jinak.

Rozhraní DOM je pro nás přístupné především pomocí globální vestavěné proměnné **document**. Ta odpovídá té stránce, kterou prohlížeč právě zobrazuje (a jejíž JavaScript je právě vykonáván). V dokumentaci lze nalézt veliké množství vlastností a funkcí, které takový **document** nabízí – na některé z nich se podíváme v nadcházející podkapitole pro koutačky. I zelenáč ovšem musí ovládat funkci **querySelector**, kterou používáme k tomu, abychom v dokumentu vyhledali konkrétní HTML prvek. Parametrem pro **querySelector** je řetězec obsahující CSS *sektor*, který známe ze světa CSS. Tam jej používáme pro stejný účel, když vybíráme HTML značky, na které se mají aplikovat daná stylová pravidla.

V první proměnné (**song**) proto bude uložena první značka **<pre>** ve stránce – náš text písničky. Druhou proměnnou (**text**) pak naplníme samotným obsahem dané HTML značky; její datový typ bude řetězec. Používáme k tomu vlastnost **textContent**, která opět pochází ze světa DOM.

Následující řádek slouží ke skrytí většiny textu:

```
song.textContent = text.substring(0, 50) + "...";
```

To je naše první opravdová *manipulace* s DOMem, tedy se stránkou! Přiřazením do vlastnosti **textContent** rozkazujeme, že je nutné HTML pozměnit nahrazením textu, který se zobrazuje uvnitř značky **<pre>** (tj. uvnitř proměnné **song**). Vypíšeme v ní prvních padesát znaků z textu (funkce **substring** vrátí podřetězec od/do zadané pozice) a k tomu ještě připojíme tři tečky – naznačení, že v těchto místech je cosi skryto. Můžeme si rovnou povšimnout použitého znaménka **+**, které v JavaScriptu funguje nejen pro aritmetiku (sčítání), ale také pro spojování řetězců. V dalších kapitolách ovšem uvidíme, že tato forma kombinace řetězců není jediná a často je vhodnější použít jiný zápis.

Následuje vytvoření tlačítka pro odkrytí zbylého textu. Seznámíme se tak s dalšími dvěma funkcemi rozhraní DOM – **createElement** pro vytvoření nového HTML prvku a **append** pro jeho vložení dovnitř stránky. V našem případě chceme tlačítko ukázat za vytečkovaným textem, tj. na konci HTML prvku **<pre>** (který máme uložený v proměnné **song**). Nově vytvořené tlačítko přiřadíme do proměnné **button**:

```
let button = document.createElement("button");
button.textContent = "zobrazit celý text";
song.append(button);
```


Poslední přísada do funkčního řešení je sice drobná, ale zásadní. Musíme provázat kliknutí na tlačítko se zobrazením zbylého textu. Ve světě klientského JavaScriptu (a víceméně i kdykoliv jindy, kdy pracujeme s uživatelskými rozhraními) se k tomu používá koncept nazvaný *programování řízené událostmi* (anglicky *event-based programming*). V našem případě to znamená následující činnosti:

1. Rozmyslet, při které události či interakci chceme nějaký JavaScript spustit.
2. Připravit si tento kód a vytvořit pro něj JS *funkci*.
3. Propojit výše uvedené pomocí DOM funkce **addEventListener**.

Těmto aktivitám odpovídají poslední řádky z řešení:

```
function showCompleteText() {
    song.textContent = text;
}
button.addEventListener("click", showCompleteText);
```

Definice funkce je velmi jednoduchá, neboť nepotřebuje žádné parametry ani návratovou hodnotu. Zvláštní pozornost ale věnujme volání **addEventListener**, které prohlížeč instruuje, aby naši funkci (předanou jako druhý parametr) vykonal, jakmile nastane kliknutí myši (řetězec **"click"** v prvním parametru) – a to celé jen pokud událost nastane na HTML prvku uloženém v proměnné **button**.

Pozor! Funkci **showCompleteText** sice definujeme, ale nikdy ji sami nevoláme. Nevíme totiž, kdy – to ví jen prohlížeč, který se od operačního systému dozví o kliknutí myši. Proto ji jen *předáváme* jako druhý parametr při volání funkce **addEventListener**. Tento zápis může být pro začátečníka matoucí, neboť vzdáleně připomíná *volání* funkce, při kterém za její jméno ještě píšeme kulaté závorky.

V JavaScriptu jsou funkce docela běžný datový typ. Pro snazší pochopení se na ně můžeme dívat jako na obyčejné proměnné; dokonce je tak lze i definovat. V našem případě bychom klidně mohli psát

```
let showCompleteText = function() {
    song.textContent = text;
}
```

Při tomto zápisu je pak předání hodnoty `showCompleteText` jako parametru do jiné funkce docela pochopitelné.

Co jsme se naučili

Po vyřešení první úlohy by měl čtenář chápat a ovládat:

- definici proměnných
- definici funkce
- nejdůležitější funkce a proměnné z rozhraní DOM: `document`, `querySelector`, `createElement`, `append` a `addEventListener`
- základy konceptu programování řízeného událostmi

Zelenáči: drobná vylepšení

Stará programátorská mantra praví, že problém bychom měli ideálně řešit ve třech krocích. Tím prvním je rychlé sestavení jednoduchého a hloupého řešení (*make it work*). Po ověření, že zhruba funguje, můžeme přistoupit k vylepšení tak, abychom pokryli všechny vstupy (*make it right*). A až na úplný závěr můžeme uvážit, zdali bychom neuměli výsledný kód ještě upravit s ohledem na výkon (*make it fast*).

Výše ukázané řešení je vystavěné přesně dle prvního kroku *make it work*. Nyní přišel čas na druhou fázi řešení, totiž *make it right*. Zkusíme kód vylepšit nikoliv ve smyslu jeho funkcionality, ale ve smyslu jeho praktického použití a znovupoužití.

Prvním nedostatkem je skutečnost, že jsme celý JavaScript napsali přímo do HTML dokumentu, mezi značky `<script>` a `</script>`. To přináší dvě nevýhody: jednak tento kód nelze znovupoužít v dalších stránkách, jednak teď náš dokument obsahuje zdrojový kód ve dvou jazycích (HTML a JS). Šikovnější by bylo, kdyby tyto dvě technologie mohly fungovat ve dvou různých souborech, a každý z nich by následně mohl měnit třeba jinak zdatný programátor. Oddělení JavaScriptu do samotného souboru je zcela běžná a přímočará operace: v HTML zůstane jen značka `<script>`, u které pomocí atributu `src` uvedeme, na které adrese se nachází soubor s JS kódem.

Další drobný nedostatek je použití čísla 50 při zkracování textu. Napříč všemi programovacími jazyky platí úmluva, že podobné veličiny ovlivňující fungování programu patří *na začátek zdrojového kódu*, abychom je nemuseli hledat, když

vysta-
ne potřeba jejich změny. Pro nás je to ideální chvíle na seznámení se s **konstantami** – to jsou hodnoty, které nelze měnit. Pracujeme s nimi podobně jako s proměnnými, jen pro jejich definici použijeme klíčové slovo **const** (namísto **let**), a jistě nás nepřekvapí, že případný pokus o jejich změnu by vyústil v chybu.

Třetí vylepšení souvisí s tím, kolik různých změn provádíme ve stránce. Náš současný kód mění HTML prvek **<pre>**, jehož obsah nejprve nahradí (změna vlastnosti **textContent**) a o pár řádků později ještě obohatí o tlačítko. Zde můžeme naše řešení trochu zkrátit a ještě vylepšit jeho čitelnost, když obě změny (text i tlačítko) provedeme naráz. Stačí jen znát správnou funkci z rozhraní DOM, která obsah **<pre>** nahradí kombinací textu, výpustky (tak se správně říká trojtečce) a tlačítka. Její jméno je **replaceChildren** a jedná se o tzv. *variadickou funkci*. To znamená, že ji lze volat s libovolným počtem parametrů. Prohlížeč je vezme jeden po druhém a vystaví z nich nový obsah daného HTML prvku.

Po aplikování tří výše uvedených vylepšení pak řešení první úlohy vypadá následovně. V HTML souboru zůstalo:

```
<h1>Mám styl Čendy</h1>
<pre>
Mezi námi je mnoho chvil
A pokusů, abych se ti zavděčil
Jenomže od tebe se člověk moc nedoví
Stále básníš o nějakém svém záhadném Čendovi
</pre>

<script src="kapitola-2.js"></script>
```

A nově vytvořený soubor **kapitola-2.js** obsahuje:

```
const LIMIT = 50;
let song = document.querySelector("pre");
let text = song.textContent;

let button = document.createElement("button");
```

```
button.textContent = "zobrazit celý text";
song.replaceChildren(text.substring(0, LIMIT), "...", button);

function showCompleteText() {
    song.textContent = text;
}

button.addEventListener("click", showCompleteText);
```

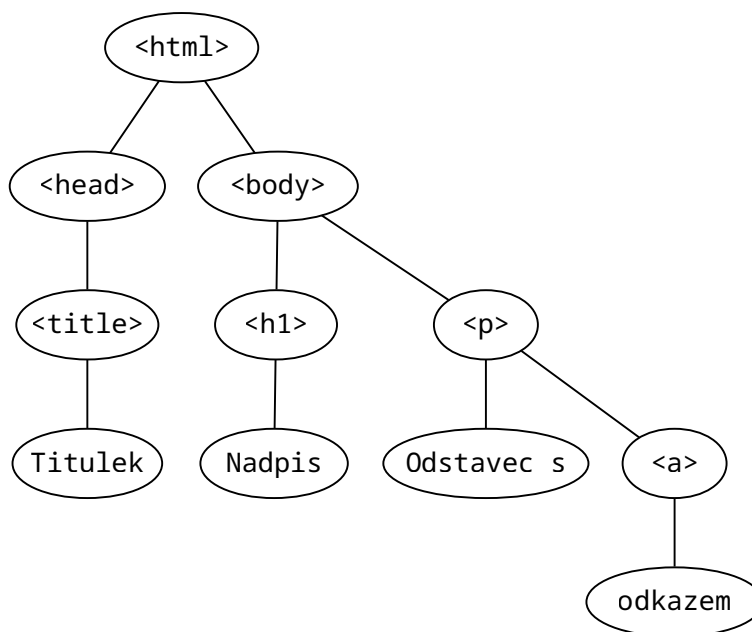
Koumáci: průzkum stromu DOMu

Zásahy ovlivňující vzhled, obsah a fungování webové stránky jsou hlavní náplní klientského JavaScriptu. Víme už, že hlavním prostředkem k tomu je sada proměnných a funkcí souhrnně nazývaná DOM. Protože takových zásahů můžeme provádět velké množství, představuje DOM pořádně objemný balík funkcionality. Jeho rozsah může být pro nováčka odrazující. Pojďme v té kupě na první pohled nesouvisejících funkcí najít nějaký smysl a říci si, na jakých pravidlech je DOM vystaven.

Klíčovou myšlenkou pro pochopení DOMu je představa, že jednotlivé HTML prvky tvoří tzv. *strom* – paměťovou strukturu, kterou si možná pamatujete z kurzů programování ve škole. Při psaní HTML nám přijde přirozené uvažovat o *zanoření* značek a to už je jen krůček od vytvoření hierarchického vztahu rodič-potomek. Pojďme se podívat na ukázkou. Mějme takovýto HTML kód:

```
<html>
  <head>
    <title> Titulek </title>
  </head>
  <body>
    <h1> Nadpis </h1>
    <p>
      Odstavec s <a>odkazem</a>
    </p>
  </body>
</html>
```

Tomu odpovídá následující strom:



Obrázek: Paměťová reprezentace HTML dokumentu

A ještě trocha terminologie:

- Paměťové stromy rostou opačným směrem, než ty v přírodě. Náš strom má jeden **kořen** (je jím značka `<html>`) a roste dolů.
- Každé jednotlivé součásti stromu se říká **uzel** (anglicky **node**).
- Každý uzel, s výjimkou kořene, má právě jednoho **rodiče**. Opačný vztah označujeme slovem **potomek**.
- Všechny uzly se společným rodičem označujeme jako jeho **přímé potomky** nebo **děti**. Slovo **potomek** tedy zahrnuje jak ty přímé, tak i jejich děti, tj. celou širokou rodinu vnoučat a pravnoučat, až k listům stromu.
- Uzly, které nemají potomky, se nazývají **listy** stromu. V HTML to jsou především tzv. **textové uzly** (to, co uživatel na stránce vidí) a potom ty HTML značky, které jsou prázdné (například obrázky nebo značky `<input>`).

- Uzly, které odpovídají HTML značkám (tedy ty, co nejsou **textové**), se nazývají **HTML prvky** (anglicky *element nodes*). Ty nás při práci s DOM zajímají nejčastěji, takže většina funkcionality v DOM se věnuje právě jim.

V této podkapitole si zkusíme shrnout funkce, které se nám hodí při práci s uvedenou stromovou reprezentací. Rozhraní DOM je s JavaScriptem historicky neodmyslitelně spojeno, takže se s námi táhne už pěknou řádku let. A protože málokdo dokáže věci vymyslet perfektně napoprvé, má za sebou i DOM celkem kontroverzní minulost. V prvních letech JavaScriptu bylo v rámci DOM k dispozici jen omezené množství funkcionality, často navržené nešikovně, a navíc implementované nekonzistentně napříč různými webovými prohlížeči. Dnes máme štěstí, že takové rozdíly už téměř neexistují a do DOMu se časem dostaly přesně ty funkce, které nám pomohou snadno vyřešit potřebnou úpravu stromu stránky.

Poslední terminologická zastávka se týká skutečnosti, že rozhraní DOM je navrženo *objektově* (dle konceptů tzv. *objektově-orientovaného programování*, zkráceně OOP). Funkce a proměnné, které přísluší nějaké části stránky, jsou dostupné pomocí operátoru tečky. Na příkladu z první řešené úlohy:

- proměnné `song.textContent` říkáme *vlastnost*, protože někomu patří (HTML prvku `<pre>`),
- funkci `button.addEventListener` říkáme *metoda*, protože někomu patří (HTML prvku `<button>`).

V kurzech objektově-orientovaného programování bývá zvykem na tuto terminologii striktně dbát. V JavaScriptu tak přísní nejsme, mimo jiné proto, že – jak si ukážeme v kapitole 8 – mezi *funkcemi* a *metodami* ve skutečnosti není téměř žádný rozdíl.

Průchod stromem

Vstupním bodem do stromu stránky je vždy proměnná **document**. V něm se pak můžeme pohybovat různými metodami a vlastnostmi:

- Metoda **document.querySelector** vrací první HTML prvek vyhovující danému selektoru. Tato metoda je dostupná i pro HTML prvky, takže můžeme psát např. **song.querySelector(...)**, a omezit tak hledání jen na potomky proměnné **song**.
- Metoda **document.querySelectorAll** (a její varianty pro všechny prvky **prvek.querySelectorAll**) vrací všechny potomky vyhovující danému selektoru.
- Vlastnost **node.parentNode** odkazuje na rodiče zadaného uzlu (uloženého v proměnné **node**).
- Vlastnost **parent.children** odpovídá všem prvkům (nikoliv textovým uzlům), které jsou přímými potomky daného rodiče. První z nich je dostupný jako **parent.firstChild**, poslední pak **parent.lastElementChild**.
- Vlastnost **parent.childNodes** odpovídá všem dětem daného rodiče (textové uzly i HTML prvky). První z nich je dostupný jako **parent.firstChild**, poslední pak **parent.lastChild**.
- Vlastnost **element.previousElementSibling** je předchozí HTML prvek ve stejném rodiči; symetricky **element.nextElementSibling** je následující HTML prvek.
- Vlastnost **element.previousSibling** je předchozí uzel; symetricky **element.nextSibling** je následující uzel.

V rozhraní DOM nalezneme i další vlastnosti a metody užitečné pro navigaci stromem, ale ty výše uvedené nám bohatě postačí.

Tvorba nových uzlů

- **document.createElement(name)** vyrobí nový HTML prvek. Je to jedináček a sirotek; dokud jej nevložíme někde do stromu, nebude mít rodiče, sourozence ani potomky.

- **document.createElementNS(ns, name)** je historickou specialitou, kterou používáme téměř výhradně k tvorbě SVG prvků. Více se o nich dočteme v bonusové čtrnácté kapitole.
- **document.createTextNode(text)** vyrobí nový textový uzel. Mohlo by se zdát, že je to docela praktická metoda, ale uvidíme, že ji téměř nikdy nepotřebujeme.

Vkládání uzlů do stromu

Níže uvedené metody fungují obdobně: vloží zadané uzly na konkrétní místo ve stromu dokumentu. Jsou variadické, takže jim můžeme předat libovolný počet parametrů. Těmi mohou být nejen HTML prvky (vzniklé například voláním **createElement** či **querySelector**), ale také obyčejné JS řetězce, které budou automaticky převedeny na textové uzly.

Pokud tímto způsobem vložíme do stromu uzel, který už předtím někde ve stromu byl, tak dojde k jeho přesunu na nové místo.

- **parent.append(child1, child2, ...)** přidá nové uzly jako potomky na konec rodiče.
- **parent.prepend(child1, child2, ...)** přidá nové uzly jako potomky na začátek rodiče (před prvního existujícího potomka, pokud nějaký je).
- **element.before(other1, other2, ...)** přidá nové uzly před daný prvek (tj. budou to jeho sourozenci).
- **element.after(other1, other2, ...)** přidá nové uzly za daný prvek (tj. budou to jeho sourozenci).

Nahrazení uzlů jinými

- **parent.replaceChildren(child1, child2, ...)** odebere všechny potomky a nahradí je novými.
- **element.replaceWith(other1, other2, ...)** nahradí daný prvek jedním či více jinými.

Odebrání uzlů

■ `element.remove()`

Ostatní

Kromě výše uvedených metod lze strom stránky měnit ještě pomocí několika dalších vlastností. My se nyní podíváme na dvě z nich, které jsou důležité jak pro *čtení*, tak pro *zápis*.

Čtením vlastnosti `element.textContent` získáme zřetězení všech textových uzlů, které jsou potomky prvku `element`. Když do této vlastnosti přiřadíme, tak zadaným textem nahradíme obsah prvku `element`. Následující dva zápisy jsou tedy identické:

```
element.textContent = "Mám styl Čendy";
element.replaceChildren("Mám styl Čendy");
```

Obdobným systémem funguje druhá vlastnost `element.innerHTML`. Jejím čtením získáme zřetězení *veškerých* potomků daného prvku, tedy nejen těch textových. HTML potomci budou převedeni na text pomocí jejich zápisu v HTML.

Přiřazení do `element.innerHTML` nám dovoluje vystavět nový podstrom z řetězce obsahujícího HTML. Stejně jako u `textContent`, původní obsah značky bude odstraněn a nahrazen novým, získaným převodem zadaného HTML na strom. Tato operace patří k těm nejsilnějším, které nám DOM nabízí – ale je nutné pamatovat, že s velkou mocí přichází také velká zodpovědnost. Neopatrné přiřazení do `innerHTML` totiž může být **zdrojem zranitelnosti typu XSS** (více se o tom dočteme v kapitole 6)!

Profíci: druhy skriptů

HTML značku `<script>` můžeme použít jedním ze dvou způsobů: buď ji naplníme kódem, nebo uvedeme atribut `src` s odkazem na JavaScriptový soubor. Chování prohlížeče ale můžeme ovlivnit ještě dalšími jejími atributy. Děláme to proto, abychom ovlivnili, kdy prohlížeč skript **stáhne**, kdy prohlížeč skript **vykoná** a co všechno je v daném skriptu **dovoleno**.

Klientský JavaScript je, stejně jako HTML a CSS, zatížen břemenem zpětné kompatibility. Je to jeho výhoda i prokletí. Výhoda proto, že náš kód napsaný před dvaceti lety dnes funguje stejně jako tehdy a do budoucna máme stejně pozitivní vyhlídky. Prokletí proto, že je velmi obtížné do jazyka přinášet novinky a změny tak, aby se tím *nic nerozbilo*. Prohlížeče kvůli tomu musí stále obsahovat implementaci všeho možného, včetně technologií, které už dávno považujeme za překonané.

Jeden mechanismus, který lze pro modernizaci použít, je zavedení *nového druhu skriptů*. Tento krok se objevil naposledy po roce 2015, kdy JavaScript prošel pořádnou aktualizací. Namísto nové HTML značky byla ovšem zavedena nová hodnota atributu **type**. Dnes tak máme dva druhy skriptů:

- **klasické skripty**, které atribut **type** nemají (nebo mají, s historickou a zbytečnou hodnotou **text/javascript**), a
- **JS moduly**, které mají atribut **type=module**.

Pojďme se na ně podívat detailněji.

Klasické skripty

Klasické skripty představují původní mechanismus spolupráce HTML a JavaScriptu. Jejich funkce a globální proměnné jsou sdíleny napříč všemi takovými skripty a není povoleno v nich používat některé moderní syntaktické prvky jazyka (zejména direktivy **import** a **export**, viz kapitolu 8).

Klasické skripty začne prohlížeč načítat, jakmile na ně v HTML narazí. Přestane přitom dělat všechno ostatní, takže pokud stažení klasického skriptu trvá dlouho, zcela to zablokuje proces načítání a vykreslení HTML stránky. Tomuto chování lze předejít přidáním atributu **async**: prohlížeč pak skript stahuje zároveň (paralelně) se zpracováváním dalšího HTML.

Klasické skripty prohlížeč vykoná, jakmile je načte. Tomuto chování lze předejít přidáním atributu **defer**: prohlížeč pak skript vykoná až poté, co zpracuje celé HTML a vystaví celý strom stránky.

Je zbytečné kombinovat atributy **async** i **defer** zároveň. Pokud má klasický skript atribut **async**, je jeho atribut **defer** ignorován (bude načten paralelně se zpracováním HTML a vykonán ihned po načtení).

JS moduly

Použitím atributu **type=module** říkáme, že chceme náš skript zpracovat modernějším mechanismem. Dostaneme tak od prohlížeče chování, které se časem ukázalo jako praktičtější (ale nemožné zavést paušálně pro všechny skripty, kvůli zpětné kompatibilitě). Všechny funkce a proměnné v JS modulu nejsou dostupné z jiných skriptů na stránce; pro spolupráci je nutné využít direktivy **export** a **import**. JS moduly jsou automaticky vykonávány ve *strikním režimu* (koumáci se o něm mohou dočíst ve čtrnácté bonusové kapitole).

JS moduly jsou automaticky zpracovávány způsobem **defer**, tj. jejich načítání probíhá zároveň se zpracováním HTML a jejich vykonávání nastane až po vytvoření stromu stránky. Pokud chceme JS modul vykonat ihned po jeho načtení (tj. potenciálně dříve, než je strom sestaven), můžeme mu přidat atribut **async**.

JS moduly s atributem **src** je nutné vydávat pomocí HTTP (tj. pomocí webového serveru, nikoliv z adresy začínající na **file://**). Jejich načítání podléhá mechanismu *same-origin* / *CORS* (viz kapitolu 5).

Příliš rychlý skript

Výše uvedené možnosti se nám hodí mimo jiné v situaci, kdy narazíme na problém způsobený skriptem, který je vykonáván příliš brzy. Typická začátečnická chyba může vypadat například takto:

```
<script>
let button = document.querySelector("button");
button.addEventListener("click", ...);
</script>

<button> Klikni! </button>
```

Poznáte, v čem je problém? Jedná se o **klasický skript**, takže jeho načtení i vykonání proběhne přesně v tom místě, kde je v HTML značka **<script>**. A až se prohlížeč pokusí vykonat metodu **document.querySelector**, ve stránce žádný prvek odpovídající selektoru nenajde. Že ho tam vidíme? My možná ano, ale prohlížeč ještě ne: při zpracování HTML se zatím tak daleko nedostal.

Pro tuto situaci existuje hned několik řešení, z kterých si můžeme vybrat to, které nám bude nejvíce vyhovovat.

1. Odložení vykonání skriptu: `<script defer>` nebo `<script type=module>`

2. Přesun skriptu v rámci HTML:

```
<button> Klikni! </button>

<script>
let button = document.querySelector("button");
button.addEventListener("click", ...);
</script>
```

3. Umístění skriptu do funkce vykonané později:

```
<script>
function go() {
  let button = document.querySelector("button");
  button.addEventListener("click", ...);
}
document.addEventListener("DOMContentLoaded", go);
</script>

<button> Klikni! <button>
```

Poslední varianta dříve patřila mezi velmi oblíbené možnosti. Použitá událost **DOMContentLoaded** odpovídá okamžiku, kdy prohlížeč dokončil načítání HTML a sestavil strom stránky. Takovéto odložení (vlastně stejné, jako atribut **defer**) je šikovné, ale nezbaví nás problematického zablokování stránky během načítání klasického skriptu. Vhodnější je proto použití prvního nebo druhého řešení.

Pole a iterace

Úloha

Kód z předchozí kapitoly se osvědčil! Rádi bychom jej nyní použili i na další stránce, kde bude naráz vypsáno několik písní. Zkrácení (a zobrazení pomocí tlačítka) je proto nutné provést na všech textech písní, které se v dokumentu objeví.

Řešení

```
<!-- kapitola-3.html -->
<h2>Mám styl Čendy</h2>
<pre class="song"> ... </pre>

<h2>Hříšné bolero</h2>
<pre class="song"> ... </pre>

<script src="kapitola-3.js"></script>
```

```
// kapitola-3.js
const LIMIT = 50;

function shortenSong(song) {
  let text = song.textContent;

  let button = document.createElement("button");
  button.textContent = "zobrazit celý text";
  song.replaceChildren(text.substring(0, LIMIT), "...", button);

  function showCompleteText() {
    song.textContent = text;
  }
}
```

```

    button.addEventListener("click", showCompleteText);
}

let songs = document.querySelectorAll(".song");
for (let i=0; i<songs.length; i++) {
    shortenSong(songs[i]);
}

```

Po zbytek knihy už budeme vždy uvažovat JavaScriptový kód oddělený od HTML. Trochu se tím komplikuje možnost si jednoduše uvedená řešení vyzkoušet, ale zase se jedná o přístup, který v praxi vídáme nejčastěji.

Výše uvedené řešení je motivováno přímočarou úvahou: chceme vzít hotový kód (z minulé kapitoly) a aplikovat jej opakovaně. Dopředu není jisté, kolikrát to bude, protože náš výsledek by měl fungovat u stránky se dvěma písněmi, stejně jako když jich tam bude padesát. V takovém případě je logické sáhnout po nějaké formě **cyklu**. JavaScript nabízí různé formy iterace (více o tom v podkapitole pro zelenáče) a my můžeme použít hned tu nejsnazší – pomocí klíčového slova **for** s iterační proměnnou. Tento zápis je známý z mnoha jiných programovacích jazyků. Používáme v něm tzv. *iterační proměnnou* (zde **i**), kterou nejprve nastavíme na nulu a následně v každé iteraci zvýšíme o jedničku (operátor **++**). To děláme tak dlouho, dokud je hodnota této proměnné menší, než zadaný limit.

Naše data jsou dostupná v proměnné **songs**, pro jejíž naplnění jsme zvolili metodu **document.querySelectorAll**. Jedná se o hodnotu, jejíž datový typ se formálně nazývá *statický NodeList*, tedy něco jako *posloupnost HTML uzlů*. V mnoha ohledech bychom takovou proměnnou mohli považovat za pole (datový typ zpravidla užívaný pro řady hodnot), ale pozor – JavaScriptové pole to není. Nelze proto použít např. funkcionální iteraci (viz dále). Nám ovšem stačí, když:

1. umíme ověřit, kolik má **songs** hodnot (pomocí vlastnosti **length**),
2. dokážeme získat konkrétní N-tou položku z této posloupnosti (pomocí operátoru hranatých závorek).

Pro potřeby této ukázky kódu jsme použili selektor **.song**, kterému vyhovují všechny HTML prvky, jejichž atribut **class** obsahuje slovo **song**. To rozhodně není jediný způsob, jak v dokumentu písně nalézt. Je to předmětem dohody mezi autorem HTML a autorem skriptu; ve stránce musí být písně zapsány takovým

způsobem, aby je pak v JavaScriptu šlo všechny *najít*. Kdybychom použili například selektor **pre**, je možné, že by kód fungoval stejně dobře. Trochu bychom se tím ale vystavili riziku, že do HTML stránky někdo následně vloží značku `<pre>`, která ovšem vůbec nebude použita k výpisu textu písničky. I takovou bychom pak naivně považovali za písničku a pokoušeli se její obsah zkrátit. Proto bude šikovnější v HTML explicitně označit (atributem **class**) jen ta správná místa, která si zaslouží zpracovat.

V každé iteraci nakonec zavoláme funkci **shortenSong**, která je jen obalem kolem kódu z minulé kapitoly. Její zodpovědností je zkrácení textu konkrétní písničky a tvorba interaktivního tlačítka. Dochází zde poprvé k situaci, kdy *definujeme funkci uvnitř jiné funkce*. To je v JavaScriptu docela běžný postup a neměl by nás překvapit (z minulé kapitoly si pamatujeme, že funkce je datový typ, takže její vznik je vlastně jen vznik stejně pojmenované proměnné). Jen si musíme dát pozor na to, abychom takové definice nevyužívali příliš hluboce zanořené: dochází tím k velkému odsazení kódu, ke snížení čitelnosti a také se tím komplikuje porozumění chování zanořených funkcí. Jejich *scope* totiž neprakticky narůstá – povíme si o tom něco v podkapitole pro profíky.

Co jsme se naučili

Po vyřešení druhé úlohy by měl čtenář chápat a ovládat:

- spolupráci mezi HTML a JS pomocí **querySelector** a **querySelectorAll**
- zápis iterace s pomocnou proměnnou
- definici funkce uvnitř funkce

Zelenáči: anonymní a arrow funkce

Doposud jsme viděli funkce definované dvěma obdobnými způsoby: kombinací klíčového slova **function** a jména. Vypadalo to zhruba takto:

```
function scitani(a, b) { return a+b }

let odcitani = function(a, b) { return a-b }
```

Byť se tyto dva zápisy funkčně *trošinku* liší, můžeme je považovat za prakticky identické. JavaScript nám ovšem dovoluje i další formy definice funkcí. Jednou z nich je tzv. *anonymní funkce*, která se od té normální liší pouze absencí jména. V ukázce s klikacím tlačítkem bychom ji mohli použít takto:

```
button.addEventListener("click", function() {
    song.textContent = text;
});
```

Chování je stejné jako dříve. Protože však funkce nemá jméno, není uložena v žádné proměnné. Nemůžeme se na ni proto odkázat ani ji vykonat. Takový zápis je praktický přesně v těch místech, kde funkci potřebujeme *jen předat* (jako např. druhý parametr `addEventListener`). Neuvedením jména ušetříme trochu místa a zároveň čtenáři naznačíme, že tato funkce je užitečná jen jako posluchač události. Nechceme ji nikam ukládat, ani sami později odnikud volat.

V roce 2015 došlo k značnému rozšíření možností JavaScriptu a společně s tím přibyla další syntaxe pro definici funkcí. Namísto slova **function** lze použít tzv. **operátor tlusté šipky** `=>` a pomocí něj definovat funkci, které se říká po anglicku *arrow function*:

```
let scitani = (a, b) => {
    return a+b;
}
```

Tento způsob vytváření funkcí má řadu odlišností od použití klíčového slova **function**. Ty nejdůležitější jsou:

- Arrow funkce používají úspornější zápis.
- Pokud má funkce přesně jeden parametr, nemusí se kolem něj psát kulaté závorky.
- Pokud funkce obsahuje jen jeden příkaz, pak se kolem těla nemusí psát složené závorky a výsledek tohoto jednoho příkazu je z funkce vrácen i bez použití klíčového slova **return**.
- Uvnitř arrow funkce nefunguje (resp. funguje odlišně) klíčové slovo **this** (viz kapitolu 8).

Arrow funkce si mezi JavaScriptovými vývojáři získaly značnou oblibu, a tak je v kódu potkáváme častěji a častěji. Za zmínku stojí, že je lze používat taktéž jako anonymní funkce. Řešení třetí kapitoly bychom proto mohli upravit pomocí anonymní arrow funkce takto:

```
button.addEventListener("click", () => song.textContent = text);
```

To je výrazné zkrácení, aniž by utrpěla čitelnost kódu.

Koumáci: druhy iterací

Cyklus **for** s iterační proměnnou je jen jednou z celé řady možností, jak v JavaScriptu opakovaně vykonávat potřebnou logiku. Jeho charakteristickým rysem je, že pro jeho konstrukci vůbec nepotřebujeme žádný pokročilejší datový typ. Kdybychom například chtěli vypsat prvních deset čísel, mohli bychom:

```
for (let i=1; i<=10; i++) {  
    console.log(i);  
}
```

Tuto vlastnost obsahuje také druhý elementární cyklus **while**:

```
let i=1;  
while (i <= 10) {  
    console.log(i);  
}
```

V obou případech stačí jedna číselná proměnná, nemusíme mít žádné pole ani jinou datovou strukturu. Jakmile máme zpracovávaná data uložená v nějaké složitější proměnné, můžeme uvažovat pokročilejší syntaxi pro iteraci. Další v řadě je cyklus **for-in**:

```
let data = {  
    name: "Jan",  
    age: 42  
}  
  
for (let p in data) {
```

```

console.log(p);          // "name", "age"
console.log(data[p])    // "Jan", 42
}

```

Pomocná proměnná **p** nabývá postupně hodnot všech *klíčů* v iterovaném objektu. Pro použití této syntaxe proto potřebujeme cokoliv, co odpovídá JavaScriptové definici *objektu*. Zde se trochu rozcházíme s terminologií objektově-orientovaného programování, kde slovo *objekt* značí *instanci třídy*. V JavaScriptu se objektem nazývá každý datový typ, který má *klíče* a jim odpovídající *hodnoty*. Spadají sem pole, funkce, množiny, třídy i jejich instance. Zejména se nám pak cyklus **for-in** hodí u proměnné **data** z této ukázky; její datový typ se formálně nazývá *objekt ex nihilo*, ale mnohem častěji mu říkáme *slovník* nebo *záznam* (anglicky *dictionary* nebo *record*).

Ve slovníku ukládáme data jako dvojice klíč-hodnota, kde klíč je vždy řetězec (i když kolem něj zpravidla nemusíme psát uvozovky) a hodnota je libovolná. O slovnících a jejich schopnostech si více povíme v osmé kapitole.

Protože JS pole je také objekt, nabízí se možnost iterovat proměnnou typu pole také pomocí syntaxe **for-in**. Jeho klíče by pak byly jednotlivé číselné indexy (hodnoty 0, 1, 2, ...). Takový postup ovšem nemůžeme vždy doporučit, protože v poli by se mohly objevit i další klíče, které naše iterace neočekává (viz kapitolu 8). Pokud jsou naše data uložena v opravdovém poli, je o něco lepší variantou **funkcionální iterace**:

```

let todo = ["cvičit", "tančit", "dožít"];

function show(task) {
  console.log("Je potřeba", task);
}

todo.forEach(show);

// varianta s anonymní funkcí
todo.forEach(function(task) {
  console.log("Je potřeba", task);
});

// varianta s anonymní arrow funkcí

```

```

todo.forEach((task) => {
  console.log("Je potřeba", task);
});

// varianta se zkrácenou anonymní arrow funkcí
todo.forEach(task => console.log("Je potřeba", task));

```

Jak to funguje? Koncept funkcionální iterace přichází z oblasti zvané *funkcionální programování*, ve kterém už podle názvu hrají prim funkce. Abychom mohli iterovat skrz pole, musíme definovat (zpravidla malinkatou) iterační funkci, která je pak automaticky **volána pro každý prvek pole**, tj. právě procházený prvek pole jí je předán jako parametr. V první ukázce vytváříme pro větší čitelnost pojmenovanou funkci **show**, v dalších ukázkách šetříme místo a funkce jsou anonymní.

Pro nás je to druhé setkání s konceptem *předávání funkce jako parametru* (první bylo u **addEventListener**). V JavaScriptu je funkcionální iterace dostupná nejen prostřednictvím metody **forEach**, ale i dalšími metodami (**map**, **filter**, **reduce**, ...). Více si o tom povíme v kapitole 6. Ideálním partnerem pro funkcionální iteraci jsou arrow funkce, jejichž úsporný zápis dobře vyrovnává funkcionální přístup, tj. *velmi časté používání minimálních funkcí*.

Fanoušci funkcionální iterace jsou v tuto chvíli možná zaskočení tím, že v řešené úloze je proměnná **songs** typu *statický NodeList*, tj. nejedná se o pole – nelze použít funkcionální iteraci. Pokud chceme, můžeme si ovšem pole vyrobit funkcí **Array.from** a naplnit ho daty z proměnné **songs**. Pak už funkcionální iteraci nic nebrání:

```

let songs = document.querySelectorAll(".song");
let songsArray = Array.from(songs);
songsArray.forEach(shortenSong);

```

Na závěr této podkapitoly si ještě ukážeme poslední iterační mechanismus, nazývaný *programovatelná iterace*. Jeho syntaxe využívá konstrukci **for-of**:

```

let todo = ["cvičit", "tančit", "dožít"];

for (let task of todo) {

```

```
console.log("Je potřeba", task);
}
```

Zápis připomíná syntaxi **for-in**, ale jedná se o velmi odlišný systém. Při této formě iterace JavaScript opakovaně volá předem domluvenou metodu objektu, který je uveden vpravo od klíčového slova **of** (této domluvě se říká *iterační protokol*). Jejím úkolem je *vrátit další položku*, která je následně přiřazena do lokální proměnné vlevo od klíčového slova **of**. V praxi to znamená, že počet cyklů a procházené hodnoty jsou řízeny výhradně objektem, který je iterován. Programátorovi to dává možnost vytvořit vlastní objekt se specializovanými schopnostmi iterace. Může to být třeba objekt zastřešující textový soubor, který postupně vrací jednotlivé řádky; nebo objekt odpovídající databázovému dotazu, který v rámci iterace vrací získaná data. Implementace iteračního protokolu je ovšem komplikovaná a dostaneme se k ní až v poslední čtrnácté kapitole.

JavaScript poskytuje implementaci iteračního protokolu pro řadu vestavěných objektů, abychom mohli cyklus **for-of** rovnou použít bez většího úsilí. Zejména je k dispozici pro všechna pole a také pro *NodeList*, který používáme v řešení úlohy třetí kapitoly:

```
let songs = document.querySelectorAll(".song");
for (let song of songs) {
  shortenSong(song);
}
```

Shrňme jednotlivé varianty iterace spolu s jejich doporučeným použitím:

- Cykly s pomocnou iterační proměnnou (**for**, **while**): tam, kde chceme sami řídit, které položky zpracujeme.
- Cyklus **for-in**: tam, kde chceme procházet klíče a/nebo hodnoty slovníku.
- Funkcionální iterace **forEach**: pouze pole, případně data, která lze na pole převést.
- Cyklus **for-of**: tam, kde potřebujeme sami naprogramovat iteraci, nebo se nám líbí možnost iterace většiny složitějších datových typů.

Profíci: scope a closure

Pojďme si vyjasnit a upřesnit otázku, ke které jsme zatím přistupovali spíš nahodile: **jaký je obor platnosti proměnných v JavaScriptu?** Intuitivně chápeme, že proměnná slouží jako označení pro hodnotu uloženou v paměti. Jak dlouho ale proměnná (a jí odpovídající paměť) existuje? Může se stát, že o hodnotu v ní uloženou přijdeme, nebo se dokonce změní na jinou?

Na úvod této podkapitoly je dobré si zopakovat, že JavaScript je jazyk, jehož paměť je spravovaná nepřímo, procesem zvaným *Garbage Collection*. Pro programátora to znamená, že je v absolutní většině případů odstíněn od manuální správy paměti. Prohlížeč ji za něj od operačního systému získá, kdykoliv je to potřeba, a naopak uvolněna zpět bude až teprve, když je to absolutně bezpečné (tj. když je jisté, že v ní již nejsou žádná data, se kterými bychom chtěli pracovat). Garbage collector (komponenta vestavěná v implementaci JavaScriptu) hlídá každou proměnnou a paměť uvolní teprve tehdy, když k proměnné nevede žádná cesta. Navíc je chytrý a paměť neuvolňuje ihned, ale až když je na to vhodná chvíle (nebo když je větší množství paměti potřeba k něčemu dalšímu). V tomto smyslu tedy nepotřebujeme žádné speciální vědomosti.

Musíme ale rozumět tomu, co znamená zjednodušené konstatování *když k proměnné nevede žádná cesta*. Při pohledu na proměnnou je totiž nutné uvážit nejen její platnost *v prostoru* (tj. kde v kódu ji můžeme používat), ale i *časovou* (kdy k takovému použití může dojít).

Prostorová platnost proměnných se anglicky nazývá *scope* a je definována podle několika pravidel ukotvených ve standardu jazyka. Nejjednodušeji je můžeme shrnout tak, že proměnná platí od své definice až po konec *bloku* (zavírací složenou závorku), ve které byla definována. Podívejme se na krátkou ukázkou:

```
let x = 3;

function scitani(a, b) {
  let sum = a+b;
  if (sum > 99) {
    let str = "Pozor s velkými čísly!";
    console.log(str);
  }
}
```

```
    return sum;
}
```

Proměnná **x** není definována v žádném bloku, takže platí od své definice až do konce souboru. Říkáme o ní, že je *globální*. Pokud je tento kód umístěn v JS modulu (viz předchozí kapitolu), tak to není zcela pravda – v takovém případě je globální jen pro kód v tomto modulu. Ostatní skripty a moduly s ní pracovat nemohou.

Proměnná **sum** je definována ve funkci, takže platí od své definice až do konce funkce **scitani**. Říkáme o ní, že je *lokální*.

Proměnná **str** je definována v bloku kódu následujícím za podmínkou. Její scope je jen do konce této podmínky; nemůžeme ji použít například tam, kde vracíme hodnotu **sum**. Toto chování je ovšem specifické pro proměnné definované klíčovým slovem **let**, které se v JavaScriptu objevily až po roce 2015. Starší kód používal klíčové slovo **var**, které vytváří scope *do konce funkce*.

Pojďme se nyní vrátit ke kódu z řešení této kapitoly. V něm pro každý nalezený text písně voláme funkci **shortenSong** definovanou takto:

```
function shortenSong(song) {
    let text = song.textContent;

    let button = document.createElement("button");
    button.textContent = "zobrazit celý text";
    song.replaceChildren(text.substring(0, LIMIT), "...", button);

    function showCompleteText() {
        song.textContent = text;
    }
    button.addEventListener("click", showCompleteText);
}
```

Za zmínku stojí proměnná **song**. Je parametrem funkce, takže se fakticky jedná o běžnou lokální proměnnou. Z minulého textu víme, že její scope je do konce funkce **shortenSong**. Všimněme si ale zajímavé odlišnosti: proměnnou **song** využíváme i ve funkci **showCompleteText**, kterou následně předáváme jako parametr pro **addEventListener**.

Tím se dostáváme k otázce časové platnosti proměnné. Neplatí totiž, že dokončením vykonávání funkce **shortenSong** může proměnná **song** zaniknout. Její scope již sice (v prostoru) skončil, ale stále se může stát, že dojde k vykonání funkce **showCompleteText**. V takovou chvíli budeme proměnnou **song** potřebovat!

Nastala důležitá situace, které se říká **uzávěra** (anglicky *closure*). To proto, že funkce **showCompleteText** do svého scope *uzavírá* proměnnou **song**. S touto proměnnou lze tedy pracovat kdekoli uvnitř dané funkce a zároveň i kdykoliv bude tuto funkci možné zavolat.

Použití uzávěr je běžné a užitečné, jistě na ně ještě narazíme. Může mít ale dramatický dopad na životní cyklus našich proměnných. Jejich uzavřením bráníme Garbage collectoru, aby uvolnil jimi zabranou paměť, a zároveň se vystavujeme riziku, že uzavřená proměnná změní svoji hodnotu dříve, než ji v uzavírací funkci využijeme. Třeba takto:

```
let buttons = document.querySelectorAll("button");
let i = 0;
while (i < buttons.length) {
  buttons[i].addEventListener("click", () => alert(i));
  i++;
}
```

V této ukázce bychom rádi, aby každé tlačítko po kliku ukázalo takové číslo, kolikáté tlačítko to je. První tlačítko nulu, druhé jedničku... Proměnná **i** je uzavřená do anonymní arrow funkce, předávané jako druhý parametr **addEventListener**. Všechny tyto malé funkce ovšem uzavírají *tu samou proměnnou*, takže po kliku na libovolné tlačítko se vždy zobrazí ta samá hodnota. Bude to proměnná **i**, která tou dobou nabývá hodnoty počtu všech tlačítek.

Co s tím? Typické řešení představuje tvorba proměnné s omezeným scope, která bude specifická pro každou iteraci cyklu. Docílit toho můžeme buď použitím cyklu **for (let i=0; i<buttons.length; i++)**, nebo dodatečnou proměnnou:

```
let buttons = document.querySelectorAll("button");
let i = 0;
while (i < buttons.length) {
  let j = i; // v každé iteraci je to nová proměnná "j"
```

```
    buttons[j].addEventListener("click", () => alert(j));  
    i++;  
}
```

A ještě jedno varování: používání uzávěr jde ruku v ruce se zanořováním definic funkcí do sebe, někdy i v několika úrovních. Dochází tím k odsazování a snižování čitelnosti kódu a také to znamená, že chování vnitřních funkcí je pak ovlivňováno (uzavřenými) hodnotami, které nemusí být při studiu kódu snadno a blízko vidět. Takovému JavaScriptu bude výrazně obtížnější porozumět, než kdyby funkce své chování ovlivňovaly výhradně pomocí parametrů.

Kontrola formuláře

Úloha

Vzpomínkový web o Karlu Gottovi je velmi populární a rádi bychom s jeho uživateli a fanoušky vstoupili do bližšího kontaktu. Na konci stránky chceme formulář, kde bude moci uživatel vložit svůj názor a zanechat na sebe telefon či e-mail, abychom se na něj mohli případně obrátit. Formulář by měl jít odeslat pouze při správně vyplněném e-mailu či telefonním čísle.

Řešení

```
<!-- kapitola-4.html -->
<form>
  <h3>Zanechejte nám vzkaz!</h3>
  <p><textarea name="text"></textarea></p>
  <p><label>
    Váš e-mail: <input type="email" name="email" />
  </label></p>
  <p><label>
    nebo telefon: <input type="tel" name="tel" />
  </label></p>
  <p><input type="submit" value="Poslat vzkaz" /></p>
</form>

<script src="kapitola-4.js"></script>
<link rel="stylesheet" href="kapitola-4.css" />
```

```
// kapitola-4.js
let form = document.querySelector("form");
let email = form.querySelector("[name=email]");
let tel = form.querySelector("[name=tel]");
```

```
const TEL_RE = /^\\+?\\d{5,12}$/;

function isEmpty(input) {
  return input.value.trim() == "";
}

function checkForm(e) {
  if (isEmpty(email) && isEmpty(tel)) {
    alert("Vyplňte e-mail nebo telefon");
    e.preventDefault();
    return;
  }

  if (!isEmpty(tel)) {
    let t = tel.value;
    if (!t.match(TEL_RE)) {
      tel.classList.add("error");
      e.preventDefault();
    }
  }
}

form.addEventListener("submit", checkForm);
```

První část řešení je opět fragment HTML dokumentu. Nebudeme se mu věnovat příliš zevrubně, protože znalost HTML u čtenáře předpokládáme a tato kniha se soustředí více na JavaScript. U formuláře pro jednoduchost nejsou specifikovány atributy **method** ani **action**, které by ve skutečnosti rozhodně chybět neměly (jejich hodnoty ovšem pro náš kód nejsou podstatné). Popisky pro jednotlivé značky `<input>` vkládáme do značek `<label>` (a tyto dva spolu propojíme zanořením). Tím jednak napomáháme přístupnosti dokumentu a jednak je pak kliknutím na popisek možno aktivovat jemu odpovídající formulářové pole.

Pro zadávání e-mailu je vhodný `<input type="email">`, který sám od sebe kontroluje korektní formát zadané adresy a zároveň na softwarové klávesnici (v mobilních zařízeních) rovnou nabízí nezbytný znak zavináče. Stejně tak pro

zadávaní telefonu se nabízí `<input type="tel">`, pro který se zobrazí klávesnice číselná. Zde ovšem žádná kontrola vstupu sama od sebe neproběhne, a tak ji budeme muset naimplementovat sami v JavaScriptu.

V tradiční úloze kontroly formulářových polí se pohybujeme na nejisté hranici mezi JavaScriptem a HTML. Atributy **required** a **pattern** nám dovolují definovat kontrolní podmínky přímo v HTML stránce bez nutnosti JavaScriptu, ale jejich schopnosti nejsou velké. HTML kontrola je prováděna jen pro konkrétní izolované pole (bez vazby na ostatní položky), jsme velmi omezeni možnostmi zobrazení textu chyby a kontrolu můžeme specifikovat jen pomocí tzv. *regulárních výrazů*. Proto ji použijeme pro e-mailové pole a to telefonní zkontrolujeme pomocí JavaScriptu.

V něm nejprve na prvních třech řádcích používáme rozhraní DOM pro získání důležitých prvků – formuláře a obou inputů. Metodě **querySelector** tentokrát předáváme složitější (atributové) selektory. Rozhodně to není jediný způsob; další možností by byl například výběr pomocí atributu **type**. Následuje naše první konstanta **TEL_RE**, ve které specifikujeme regulární výraz pro telefonní číslo. Jedná se o jakýsi *vzor* nebo *šablonu*, která speciálními znaky popisuje, jak má vypadat platná hodnota. Za zmínku stojí, že regulární výrazy mají v JavaScriptu vlastní datový typ a proměnné tohoto typu vznikají zápisem mezi dvě dopředná lomítka. Ve složitějších případech je můžeme vyrábět také funkcí **RegExp()**.

Jazyk a celý koncept regulárních výrazů převyšuje rozsah této knihy, takže si jen v rychlosti vyložíme části našeho jednoduchého výrazu:

- Znaky **^** a **\$** na začátku resp. konci výrazu říkají, že tomuto vzoru musí vyhovovat celý text kontrolovaného pole, tedy nikoliv jen nějaká podmnožina (kolem které by pak mohly být nesouvisející neplatné znaky).
- Zápis **\+?** určuje, že text smí začínat jedním znakem plus (v telefonním čísle jde o tzv. mezinárodní volací kód).
- Zápis **\d{5,12}** určuje, že zbytek textu má obsahovat posloupnost pěti až dvanácti číslic.

Nejedná se o univerzálně spolehlivou kontrolu telefonního čísla, ale spíš o ilustraci toho, jak bychom mohli zhruba postupovat. Tuto konstantu použijeme hned za chvíli, jakmile si nachystáme kontrolní funkci.

V kódu se nám hodí funkce dvě: jedna pro ověření prázdnosti formulářového pole (**isEmpty**) a jedna pro celou kontrolu formuláře (**checkForm**). Hlavní kontrolní funkci pak předáme jako parametr do **addEventListener**, neboť kontrolu chceme provést až v důsledku nějaké události. V této úloze se jako událost nabízí **submit**, tedy okamžik, kdy se uživatel chystá formulář odeslat. Tato událost je vázána na HTML formulář, a tak funkci **addEventListener** voláme jako metodu proměnné **form**, do které jsme formulář přiřadili výše.

Zbývá nastudovat tělo funkce **checkForm**. Obsahuje dvě kontroly, které odpovídají zadání úlohy. První kontrola prostě ověří, že bylo vyplněno alespoň jedno pole. Využíváme zde pomocné funkce **isEmpty**, která z předaného inputu vybere vyplněný text, odstraní z něj případné přebytečné mezery na začátku a konci (metoda **trim()**) a ověří, zda něco zbylo. Zvídavý čtenář si může povšimnout, že v porovnání se zadáním používáme obrácenou logiku: namísto testu *je nějaké pole vyplněné?* se ptáme *jsou obě pole prázdná?* Vlastně tedy ověřujeme, zdali je formulář vyplněn špatně. To nám dovoluje použít programátorský koncept nazvaný anglicky *return early*: chceme přestat vykonávat funkci jakmile ověříme, že to nemá smysl. Při nesprávném vyplnění zakončíme kód voláním **e.preventDefault()**, k jehož vysvětlení se dostaneme za malý okamžik.

Druhý test využívá regulárního výrazu z konstanty **TEL_RE**, a pokud mu zadané telefonní číslo neodpovídá, nastává opět chybový stav. Tentokrát neukážeme uživateli nehezky **alert**, ale pokusíme se problém naznačit vizuálně – například vyplněním políčka červenou barvou. Z JavaScriptu bychom sice pomocí DOM mohli přímo ovlivňovat vzhledové atributy daného prvku, ale takový postup je nepraktický. Těžko by se nám hluboko uvnitř JS souborů hledalo, kde a jak se kterému poli nastavuje jaká barva, takže pro definici vzhledu upřednostňujeme jazyk CSS. Proto raději volíme přístup, kdy JavaScriptem měníme hodnotu atributu **class**, který v HTML slouží právě k tomuto účelu. Označujeme pomocí něj prvky, které se svými vlastnostmi nějak odlišují od ostatních, a chceme na ně aplikovat specifická stylová pravidla. Fanoušci lososové barvy pak mohou doplnit například následující CSS:

```
/* kapitola-4.css */
.error { background-color: salmon }
```

Z JavaScriptu ovšem nechceme nastavit atribut **class** na novou hodnotu **error**, neboť tento atribut může mít hodnot více (oddělených mezerami) a náš kód dopředu nemusí vědět, zdali a proč už tam nějaké hodnoty jsou. Je proto

praktičtější postupovat *defenzivněji* a k existující hodnotě `class` jen něco nového přidat. K takovému účelu nejlépe slouží objekt `classList`, jehož metody dovolují k atributu `class` přidávat další hodnoty či odebírat existující.

Poslední otázkou k vyřešení je metoda `e.preventDefault()` volaná v obou dílčích kontrolách. Jedná se o tzv. *metodu objektu události*, tedy logiku, která nám je k dispozici jen ve speciálních chvílkách, konkrétně v průběhu zpracování nějaké události. Vzpomeňme si, že náš současný kód (funkce `checkForm`) byl naplánován k vykonání teprve tehdy, když se uživatel pokusí odeslat formulář. Jakmile tato situace nastane, je v prohlížeči vytvořen *objekt události*, který popisuje skutečnosti pro tuto událost relevantní. Při vykonávání všech *posluchačů* dané události je pak objekt události každému posluchači předán jako parametr.

V objektu události nalezneme řadu užitečných informací a také několik metod. Ta nejdůležitější je `preventDefault`, nemá žádné parametry, a pokud ji kterýkoliv posluchač vykoná, žádá tím prohlížeč, aby **na tuto událost po vykonání posluchačů dále nereagoval**. To je smysluplné jen u takových událostí, které představují nějakou aktivitu pro prohlížeč samotný: může jít o událost kliknutí na odkaz (způsobí navigaci), stisk klávesy ve formulářovém poli (způsobí vložení znaku) nebo kliknutí na odesílací tlačítko (způsobí odeslání formuláře). Když v našem kódu kontrola selže, voláním `e.preventDefault()` zařídíme, aby nedošlo k odeslání formuláře s neplatnými daty.

Co jsme se naučili

Po vyřešení třetí úlohy by měl čtenář chápat a ovládat:

- zamezení zpracování události metodou `preventDefault`
- použití regulárního výrazu pro kontrolu textu
- použití rozhraní `classList` pro snadnou úpravu HTML atributu `class`

Zelenáči: další druhy událostí

Svět DOM událostí je pestrý a nabízí nám nástroje k tvorbě uživatelsky přívětivých stránek a aplikací. V řešených úlohách jsme se zatím setkali se dvěma událostmi a jejich původci:

- Událost **click** odpovídající kliknutí myší (či prstem, stylusem, ...) na libovolný HTML prvek.
- Událost **submit** odpovídající pokusu o odeslání formuláře, nastávající jen na HTML formulářích. Tato událost může být vyvolána různými způsoby; zejména to je kliknutí na odesílací tlačítko nebo stisk klávesy Enter, pokud je aktivní některé formulářové pole.

V dokumentaci rozhraní DOM můžeme nalézt desítky dalších druhů událostí. V kontextu kontroly formulářových polí připadají v úvahu například tyto:

- Událost **focus** nastává na HTML prvku `<input>`, jakmile tento začne být aktivní (uživatel do něj klikne, nebo se do něj přesune klávesou Tab). Symetricky s tím událost **blur** odpovídá ztrátě aktivity formulářového pole.
- Na formulářových polích vznikají události související s klávesnicí. Při stisku klávesy je to **keydown**, při puštění následně **keyup**.
- Pokud nás nezajímá, jakým způsobem ke změně formulářového pole došlo (klávesnicí, myší, vložením ze schránky, ...), můžeme použít událost **input**, která odpovídá libovolné úpravě daného pole.

Pro různé scénáře volíme různé události či jejich kombinace. Abychom vzorové řešení vylepšili, můžeme políčko pro telefonní číslo zkontrolovat dříve, než se uživatel pokusí formulář odeslat. Stisk každé klávesy (případně jiná změna hodnoty) je ovšem zbytečně agresivní, neboť bychom pak pole kontrolovali už od prvního zadaného znaku (a považovali ho za nesprávně vyplněné, i když jej uživatel plánuje vyplnit správně). Pro tento scénář je ideální událost **blur**, tedy opuštění aktivního pole.

K tomu se nám bude hodit nová funkce a nový posluchač události:

```
function checkPhone() {
}
tel.addEventListener("blur", checkPhone);
```

Budeme v této funkci chtít volat **preventDefault**? Nikoliv, protože na tuto událost prohlížeč sám nijak nereaguje, takže mu nemáme co zakazovat. Proto bychom se patrně mohli obejít bez parametru **e**, tj. bez objektu události (tak, jako v druhé a třetí kapitole).

Tato kontrolní funkce nalezne telefonní pole v jednom ze tří stavů:

1. prázdné → není potřeba kontrolovat, resp. pole neobsahuje chybu
2. vyplněné špatně → je nutno označit jako chybné
3. vyplněné správně → je nutno neoznačovat jako chybné

První implementace by mohla vypadat takto:

```
function checkPhone() {
  if (isEmpty(tel)) {
    tel.classList.remove("error");
  } else {
    let t = tel.value;
    if (t.match(TEL_RE)) {
      tel.classList.remove("error");
    } else {
      tel.classList.add("error");
    }
  }
}
```

Kód je ovšem zbytečně košatý, zanořený a není snadné z něj rychle a snadno odhadnout, za jakých podmínek je pole považováno za špatně či správně vyplněné. Můžeme jej snadno zjednodušit za použití dvou triků:

- Operátor *nebo* (znaky `||`) platí, jen když je splněna libovolná ze dvou podmínek po jeho stranách. Tím dokážeme snadno popsat podmínku ze zadání, že *telefonní pole je správné, když je prázdné nebo vyplněné dle regulárního výrazu*.
- Metoda `classList.toggle()` do atributu `class` přidá danou hodnotu, pokud je její druhý parametr pravdivý. V opačném případě danou hodnotu z `class` odebere.

Když už jsme v úpravách funkce **checkPhone**, všimneme si také, že její chování je závislé na proměnné **tel**, jejíž hodnota je ve funkci uzavřena (tento jev je detailněji vysvětlen v podkapitole pro profíky v předchozí kapitole). Zde je prostor pro zvýšení čitelnosti. V objektu události (který jsme plánovali ignorovat) je totiž mimo jiné obsažen také HTML prvek, na kterém událost nastala. Dozvíme se jej pomocí vlastnosti **e.target**. Proto můžeme vrátit parametr **e**, a tím funkci explicitně dodat veškerá data, která potřebuje:

```
function checkPhone(e) {
  let tel = e.target;
  let t = tel.value;
  let isOk = isEmpty(tel) || t.match(TEL_RE);
  tel.classList.toggle("error", !isOk);
}
tel.addEventListener("blur", checkPhone);
```

Tato funkce **checkPhone** je kratší, bez zanoření, a také ji lze použít pro zpracování více telefonních polí naráz! Ale ještě ji musíme upravit jednou.

Pokud logiku kontroly telefonního pole přesuneme do **checkPhone**, znamená to, že ji budeme chtít volat i ve chvíli kontroly celého formuláře (funkce **checkForm**). Ale naše současná **checkPhone** není dobře připravena k zavolání z jiné funkce: nemá návratovou hodnotu a jako parametr očekává objekt události vyvolané na telefonním inputu. Přidáme proto funkci návratovou hodnotu a změníme její parametrizaci. Namísto objektu události jí předáme rovnou input, se kterým má pracovat. Tím zůstane zachována její obecnost a zároveň ji budeme moci použít nezávisle na tom, jaká událost probíhá.

Následující rošáda je v JavaScriptu velmi běžná. Máme konkrétní představu o tvaru naší funkce, ale zároveň ji chceme předat jako posluchač, a proto musíme respektovat parametr s objektem události. Vytvoříme si proto jako posluchač malinkou **anonymní arrow funkci**, která tu opravdovou zavolá s upraveným parametrem:

```
function checkPhone(tel) {
  let t = tel.value;
  let isOk = isEmpty(tel) || t.match(TEL_RE);
  tel.classList.toggle("error", !isOk);
  return isOk;
```



```
}
tel.addEventListener("blur", e => checkPhone(e.target));
```

Všimněme si, že v ukázce výše definujeme dvě různé funkce. Najdete je?

Zbývá upravit zbytek kódu tak, abychom uvnitř kontroly celého formuláře mohli znovupoužít naši dílčí kontrolní funkci **checkPhone**. Celé vylepšené řešení této kapitoly pak vypadá takto:

```
let form = document.querySelector("form");
let email = form.querySelector("[name=email]");
let tel = form.querySelector("[name=tel]");

const TEL_RE = /^\\+?\\d{5,12}$/;

function isEmpty(input) {
  return input.value.trim() == "";
}

function checkPhone(tel) {
  let t = tel.value;
  let isOk = isEmpty(tel) || t.match(TEL_RE);
  tel.classList.toggle("error", !isOk);
  return isOk;
}

tel.addEventListener("blur", e => checkPhone(e.target));

function checkForm(e) {
  if (isEmpty(email) && isEmpty(tel)) {
    alert("Vyplňte e-mail nebo telefon");
    e.preventDefault();
    return;
  }

  if (!checkPhone(tel)) {
    e.preventDefault();
  }
}
```

```
}  
form.addEventListener("submit", checkForm);
```

Koumáci: výjimky

Koncept práce s výjimkami (anglicky *exceptions*) je součástí JavaScriptu po velmi dlouhou dobu (ve standardu se objevuje od konce roku 1999). Je postavený stejným způsobem jako v celé řadě dalších jazyků – C++, C#, Java, PHP, Python, Ruby a podobně. Čtenáři s výjimkami již dříve seznámení tak naleznou základní syntaktické prvky (zejména klíčová slova **try**, **catch** a **throw**) v podobě, kterou znají. Pro ostatní je určena tato podkapitola, ve které se naučíme nejen zpracovávat výjimky vyvolané mimo vlastní kód, ale i vytvářet a využívat výjimky vlastní.

Na náš kód se často díváme jako na množství funkcí, které se navzájem volají. Nezbytnou součástí volání je taktéž předávání hodnot: dovnitř funkce pomocí parametrů, ven z funkce pomocí návratové hodnoty klíčovým slovem **return**. Celá tato soustava funguje dobře za předpokladu, že při volání funkce dochází ke shodě očekávání volaného (dostane parametry, které potřebuje) a volajícího (z volané funkce jsou vrácena správná data). Problém nastane ve chvíli, kdy některá funkce není schopna požadavek splnit: buď dostala data, se kterými si neumí poradit, nebo jí nějaká vnitřní překážka zabraňuje činnost vykonat a potřebná data vrátit. Jak má taková funkce neschopnost *splnit úkol* dát najevo?

Tradičním mechanismem je nechat funkci, aby vrátila nějakou specifickou hodnotu, ze které volající pozná, že funkce nedokázala vypočítat a vrátit to, co měla. Takové řešení funguje uspokojivě v celé řadě programovacích jazyků, ale přináší jisté nepohodlí. Na první pohled v něm vidíme tyto nedostatky:

1. Podle čeho z návratové hodnoty poznat, že se jedná o chybový stav? Pokud má například funkce za úkol provést výpočet a vrátit (libovolné) číslo, tak nelze použít žádnou konkrétní *chybovou hodnotu* (nulu, minus jedničku), neboť by se mohlo jednat o korektní výsledek.
2. Pokud funkce může kromě normálního výsledku vrátit i chybovou hodnotu, znamená to, že každé místo v kódu, kde je volána, bude muset umět na tuto chybovou hodnotu zareagovat. Pokud funkci voláme z více míst, budeme muset každé takové místo vybavit speciální logikou na zpracování chyby.

Odpovědí na výše uvedené problémy je systém výjimek. Ten staví na myšlence, že chybový stav ve funkci není signalizován konkrétní vrácenou hodnotou, ale okamžitým pozastavením vykonávaného kódu, po kterém následuje vyhledání nějaké vhodné komponenty, která je ochotna na tento stav zareagovat a převzít řízení. Konkrétně, pokud naše vlastní funkce potřebuje signalizovat neschopnost dokončit zadanou práci, použije klíčové slovo **throw**. Tuto situaci, nazývanou *vyvolaná výjimka* nebo *vyhozená výjimka*, můžeme následně zpracovat, pokud je právě vykonávaný kód uvnitř bloku označeného klíčovým slovem **try**. Dojde k posunu na odpovídající blok kódu označený slovem **catch** (této části říkáme *chycená výjimka*) a pokračuje se ve vykonávání. Pojďme se na to podívat na příkladu:

```
let tel = form.querySelector("[name=tel]");
const TEL_RE = /^\\+?\\d{5,12}$/;

function isEmpty(input) {
  if (!input) { throw new Error("No input available"); }
  return input.value.trim() == "";
}

function checkPhone(tel) {
  let t = tel.value;
  let isOk = isEmpty(tel) || t.match(TEL_RE);
  tel.classList.toggle("error", !isOk);
  return isOk;
}

try {
  let isOk = checkPhone(tel);
  console.log(isOk);
} catch (e) {
  console.log(e.message);
}
```

Úplně na konci ukázky je vidět uzavření volání funkce **checkPhone** do konstrukce **try-catch**. Odpovídá to intuitivní představě *budeme vykonávat první kus kódu, a když se to nepodaří, budeme pokračovat jiným kusem kódu*. Pokud by kupříkladu

uvnitř funkce **checkPhone** došlo k vyvolání výjimky, řádek **console.log(isOk)** se nevykoná a namísto toho se vypíše **console.log(e)**. V proměnné **e** pak nalezneme tu hodnotu, která byla vyvolána (tj. výjimku).

Co ještě stojí za povšimnutí:

- Blok **try-catch** může zachytit mnoho různých výjimek. V našem kódu vyvoláváme jen jednu (uvnitř funkce **isEmpty**), ale interpret JavaScriptu sám od sebe dokáže výjimek vyvolat celou řadu. Pokud například v dokumentu nebude žádný prvek vyhovující selektoru, bude proměnná **tel** nabývat hodnoty **null**. Výjimka tak bude vyvolána již na prvním řádku funkce **checkPhone** (neboť hodnota **null** nemá vlastnost **value**). Obdobně pokud použitému selektoru bude vyhovovat HTML prvek, který není **<input>**, nastane další výjimka při přístupu k **input.value.trim** (neboť **input.value** bude **undefined**, a tím pádem nebude mít vlastnost **trim**).
- Chycení výjimky lze provést *daleko* od místa, kde byla vyvolána. V této ukázce je do bloku **try-catch** zabalena funkce **checkPhone**, ale vznik výjimek očekáváme až teprve ve funkci **isEmpty**. Z toho je vidět, že zodpovědnost za zpracování výjimky neleží na tom, kdo funkci přímo volá (porovnejme s druhou výhradou vůči návratovým hodnotám výše).
- Hodnotu vyvolané výjimky píšeme vpravo od klíčového slova **throw**. Je pak k dispozici jako lokální proměnná v bloku **catch** a používáme ji jako nositel informace o tom, jaký problém nastal. Smíme použít zcela libovolnou JavaScriptovou hodnotu, ale bývá obvyklé používat objekty typu **Error** (jako na ukázce) nebo jejich podtřídy. Tyto jsou vybaveny textovým popisem problému, které je dostupný ve vlastnosti **message**.

Používání výjimek může být užitečné, nicméně pro četnost a konkrétní realizaci nejsou stanovena žádná pravidla. Je tak na naší vlastní zkušenosti a úvaze, které všechny situace považujeme za vhodné kandidáty pro obalení do bloku **try-catch**. Při kontrole formulářových polí může nastat řada situací, při kterých chceme odeslání formuláře zamezit. Jednou z alternativ k současnému řešení může být právě vyvolávání výjimek pro nesprávně vyplněná pole. Kód z právě řešené úlohy by s využitím výjimek vypadal třeba takto:

```
let form = document.querySelector("form");
let email = form.querySelector("[name=email]");
let tel = form.querySelector("[name=tel]");
```

```
const TEL_RE = /^+\?\d{5,12}$/;

function checkEmpty(email, tel) {
  let emailValue = email.value.trim();
  let telValue = tel.value.trim();
  if (emailValue == "" && telValue == "") {
    throw new Error("Vyplňte e-mail nebo telefon");
  }
}

function checkPhone(input) {
  if (!input.value.match(TEL_RE)) {
    throw new Error("Špatně vyplněný telefon");
  }
}

tel.addEventListener("blur", e => {
  let tel = e.target;
  tel.classList.remove("error");
  try {
    checkPhone(e.target);
  } catch (err) {
    tel.classList.add("error");
  }
});

form.addEventListener("submit", e => {
  try {
    checkEmpty(email, tel);
    checkPhone(tel);
  } catch (err) {
    e.preventDefault();
  }
});
```

Poslední část práce s výjimkami je klíčové slovo **finally**, jehož použití je volitelné. Pokud chceme, můžeme k částem **try** a **catch** přidat ještě třetí blok kódu uvozený slovem **finally**. Tento bude vykonán v obou případech, tj. buď po úspěšném provedení části **try**, nebo po vykonání části **catch** v rámci zachycení výjimky. Blok **finally** se tak podobá kódu, který následuje mimo sekci **try-catch**, s jedním důležitým rozdílem: bude proveden, i pokud se v **try** nebo **catch** objeví klíčové slovo **return**.

Představme si například formulář s odesílacím tlačítkem. Pokud by jeho kontrola trvala dlouho (více o tom v následující kapitole), nervózní uživatel by mohl na tlačítko klikat opakovaně. Proto budeme chtít po odeslání tlačítko zneaktivnit a teprve po dokončení kontroly jej opět aktivovat. K tomu můžeme využít blok **finally**:

```
function onSubmit(e) {
  e.preventDefault();
  let button = e.target.querySelector("[type=submit]");
  button.disabled = true;
  try {
    checkForm(e.target);
  } catch (e) {
    alert(e.message);
    return;
  } finally {
    button.disabled = false;
  }

  sendDataToServer();
}
```

V této ukázce bychom v rámci odeslání formuláře rádi poslali získaná data na server vlastní funkcí. Je jasné, že to budeme dělat, jen pokud při kontrole nenastane výjimka – proto je blok **catch** ukončen příkazem **return**. Ale i v takové situaci potřebujeme po dokončení kontroly tlačítko učinit aktivním. Proto tento kód umístíme do bloku **finally** a máme zajištěno, že bude vykonán za všech okolností.

Profíci: zpožděná kontrola při psaní

Doposud jsme v této kapitole potkali dvě události vhodné pro kontrolu formuláře: **submit** (pokus o odeslání formuláře) a **blur** (opuštění dříve aktivního pole). Pojdme se podívat, zdali by v některé situaci dávalo smysl provádět kontrolu ještě dříve. Proč? Aby měl uživatel pokud možno co nejrychlejší zpětnou vazbu ohledně správnosti vyplnění.

V podkapitole pro začátečníky je zmíněna událost **input**, která vzniká při každé změně hodnoty formulářového pole. Je jasné, že tímto způsobem můžeme formulář kontrolovat opravdu s minimálním zpožděním za uživatelem. Zároveň je to ale příliš agresivní forma kontroly. Na vzorovém políčku s telefonním číslem vidíme, že pokud bychom jej ověřovali takto často, budeme uživatele zbytečně stresovat, i pokud plánuje vyplnit docela správný telefon. Do prázdného pole musí totiž napsat alespoň pět číslic, než bude naše kontrolní funkce spokojena, takže první čtyři budou považovány za chybu. Teprve po pátém stisku klávesy začneme pole považovat za korektně vyplněné. Pokud se uživatel podívá na monitor dříve, uvidí informaci o chybě a bude po právu zmatený.

Pro takový scénář neexistuje žádná DOM událost, která by odpovídala situaci *uživatel již napsal hodnotu a neplánuje ji dále měnit*. S trochou JavaScriptu ji ale dokážeme poznat sami. Stačí naši úlohu trochu přeformulovat: potřebujeme poznat, kdy se hodnota pole už nějakou dobu nezměnila. Jakmile tato zůstala stejná po nějaký čas, můžeme předpokládat, že uživatel již dopsal (a je čas na kontrolu). Takové strategii se anglicky říká **debounce** a můžeme se s ní setkat i mimo prostředí webových stránek.

Jak na to? V klientském JavaScriptu existuje funkce **setTimeout** dovolující vykonat zadaný kód po uplynutí nějakého časového intervalu. První návrh řešení by tedy mohl vypadat takto:

```
const DEBOUNCE = 500;

function checkPhone() {
  // stejná, jako v podkapitole pro začátečníky
}

function onInput() {
  setTimeout(checkInput, DEBOUNCE);
}
```

```
}
tel.addEventListener("input", onInput);
```

Funkce **setTimeout** funguje podobně jako **addEventListener**. Musíme jí dát nějakou funkci k odloženému spuštění a také čas, za který má být vykonána. Čas jsme uložili do konstanty **DEBOUNCE** (udává se v milisekundách). Kód výše tedy nebude kontrolovat pole ihned po každém stisku – kontrolní funkce se spustí až půl vteřiny od změny políčka.

Zkušené oko profíka ovšem vidí, že takhle to fungovat rozhodně nebude. Jakmile uživatel začne psát, každý stisk klávesy *naplánuje* jedno vykonání kontrolní funkce. Tento proces sice začne se zpožděním půl vteřiny, ale poté se vrátíme k původnímu problému, kdy uživatel během psaní vidí pole označené jako špatně vyplněné. Nesplnili jsme totiž podmínku, že kontrolu provedeme, jen pokud se hodnota od okamžiku naplánování nezměnila.

Odpovědí na tento problém je funkce **clearTimeout**. Pomocí ní můžeme vzít zpět dříve naplánované zpožděné vykonání funkce. Naše strategie tedy bude následující:

1. při změně pole naplánujeme kontrolu;
2. pokud v čase mezi změnou a kontrolou nastane další změna, první kontrolu zrušíme a naplánujeme další.

Abychom mohli funkci **clearTimeout** použít, musíme si nejprve uložit návratovou hodnotu dřívějšího volání **setTimeout**. Ta slouží jako jakýsi klíč, pomocí kterého můžeme odložené vykonání zrušit. Zmiňovaná návratová hodnota je číslo, ale protože ji použijeme jen pro předání do **clearTimeout**, tak nás její datový typ vlastně vůbec nemusí zajímat. Pojďme nyní kód rozšířit o rušení dříve naplánovaných kontrol:

```
const DEBOUNCE = 500;
let timeout; // identifikace právě naplánované kontroly

function onInput() {
  if (timeout) {
    clearTimeout(timeout);
```



```

    }
    timeout = setTimeout(checkInput, DEBOUNCE);
  }

tel.addEventListener("input", onInput);

```

Proměnná **timeout** zde hraje roli hlídacího psa, který zařídí, že naplánovanou kontrolu můžeme mít maximálně jednu. Pokud již nějakou máme (podmínka **if**), nejdříve ji zrušíme a teprve poté naplánujeme další.

Na konci podkapitoly pojďme ještě vyjasnit případné otázky, které mohou profíka při pohledu na tento kód napadnout.

- Pokud je návratová hodnota **setTimeout** číslo, může to být i nula? Pokud ano, pak je náš kód nekorektní. Kdybychom od funkce **setTimeout** dostali nulu, nebude při příští události splněna podmínka a my naplánujeme další kontrolu bez zrušení té předchozí. Naštěstí se to nestane, neboť **setTimeout** vrací jen kladná čísla.
- Představme si scénář, kdy uživatel stiskne klávesu, počká jednu vteřinu a pak ji stiskne podruhé. Náš kód naplánuje kontrolu (proměnná **timeout** se nastaví), za půl vteřiny ji vykoná (to je dobře) a po dalším stisku ... dojde k volání **clearTimeout**? Ano, protože **timeout** v sobě stále drží identifikaci již vykonané zpožděné kontroly. Nic špatného se nestane, neboť funkce **clearTimeout** u již proběhlého timeoutu prostě nic neudělá. Je to nicméně zbytečné volání a programátorská intuice nám možná radí, že bychom mu měli předejít. Řešením by pak bylo ve funkci **checkInput** proměnnou **timeout** vyprázdnit (nastavit třeba na nulu nebo **undefined**).
- Jak by to vypadalo, kdybychom tímto způsobem chtěli kontrolovat více položek? Narážíme na skutečnost, že **timeout** je globální proměnná, takže kdybychom s ní chtěli pracovat z více posluchačů, mohlo by dojít k problémům. Co s tím? Nabízí se dvě hlavní kategorie řešení. Buď timeouty ukládat do složitější datové struktury (pole či slovníku), nebo pro každou formulářovou položku vyrobit vlastní **timeout** pomocí uzávěry. Kód by pak mohl vypadat zhruba takto:

```

const DEBOUNCE = 500;

function debounceInput(tel) {

```

```
let timeout;

function onInput() {
  if (timeout) {
    clearTimeout(timeout);
  }
  timeout = setTimeout(checkInput, DEBOUNCE);
}

tel.addEventListener("input", onInput);
}

debounceInput(tel1);
debounceInput(tel2);
debounceInput(tel3);
```

HTTP na pozadí

Úloha

Pro hudební fanoušky nyní chystáme webovou službu, ve které budou moci sdílet své kulturní zážitky. Bude přístupná jen registrovaným uživatelům, kterých očekáváme velké množství. Při registraci si vyberou uživatelské jméno a heslo; těmito údaji se budou následně přihlašovat. S ohledem na plánovaný zájem je nutné uživatele v průběhu procesu registrace zavčas varovat, pokud je jimi zvolené uživatelské jméno již zabráno někým jiným. Za tímto účelem jsme nechali vytvořit **backendové JSON HTTP API**, které je nutno využít.

Řešení

U této úlohy si napřed musíme ujasnit, co se po nás vlastně chce. V textu zadání se objevuje řada zkratk a méně zkušený čtenář by se mohl zaleknout. Naštěstí nejde o nic složitého.

Podstatou úlohy je opět kontrola formuláře. Tentokrát ale není možné jen pomocí JavaScriptu rozhodnout, zdali je požadované uživatelské jméno dostupné. K tomu bychom museli v rámci webové stránky znát veškerá zaregistrovaná jména, což určitě nechceme (dle slov zadání jich bude veliké množství) ani nemůžeme (tím bychom je prozradili každému kolemjdoucímu). Proto nezbyvá, než testované uživatelské jméno poslat po síti na server, který má k dispozici databázi uživatelů a o existenci může rozhodnout. Co víc, pokud bude jméno zabrané, dokáže vyprodukovat dostupnou alternativu.

Komunikace po síti je ve světě webových stránek v 99 % případů realizována protokolem HTTP. Se serverovou stranou tedy budeme komunikovat tímto způsobem; v zadání se dále píše, že data dostaneme ve formátu JSON. To je jednoduchý, užitečný a poměrně praktický způsob, jak zapsat libovolně komplexní data tak, aby se snadno zpracovávala v JavaScriptu (resp. dnes už víceméně v libovolném dalším jazyce). Poslední použitá zkratka je API – anglicky *Application Programming Interface*. Tím se myslí, že existuje dohoda o tom, jak

mají vypadat data přenášená od klienta na server a stejně tak jak má vypadat odpověď od serveru. Při opravdovém programování bychom se na konkrétní tvar API šli zeptat backendového programátora (případně si jej přečetli v dokumentaci, nebo jej dokonce sami navrhli). V této testovací úloze budiž naše API takovéto:

1. data odesílaná na server budou realizována HTTP požadavkem poslaným na adresu `/username-check?username=...`
2. pokud je uživatelské jméno volné k registraci, server odpoví daty ve formátu JSON:

```
{
  "available": true
}
```

3. pokud je uživatelské jméno zabrané, server odpoví daty ve formátu JSON:

```
{
  "available": false,
  "suggested": "navrzene-jmeno"
}
```

S takto upřesněným zadáním se již můžeme podívat, jak lze danou úlohu vzorově vyřešit.

```
<!-- kapitola-5.html -->
<form>
  <h3>Registrujte se!</h3>
  <p><label>
    Uživatelské jméno: <input type="text" name="username" />
  </label></p>
  <p><label>
    Heslo: <input type="password" name="password" />
  </label></p>
  <p><input type="submit" value="Vytvořit účet" /></p>
</form>

<script src="kapitola-5.js"></script>
```

```
// kapitola-5.js
function getError() {
    return username.labels[0].querySelector(".error");
}

function hideError() {
    let error = getError();
    if (error) { error.remove(); }
}

function showError(suggested) {
    hideError();
    let error = document.createElement("div");
    error.className = "error";
    error.textContent = `Jméno je zabráno, zkuste třeba "${suggested}"`;
    username.labels[0].append(error);
}

function onLoad(e) {
    let data = e.target.response;
    if (data.available) {
        hideError();
    } else {
        showError(data.suggested);
    }
}

function checkUsername() {
    let xhr = new XMLHttpRequest();
    let u = encodeURIComponent(username.value);
    let url = `/check-username?username=${u}`;
    xhr.responseType = "json";
    xhr.open("GET", url);
    xhr.send();
    xhr.addEventListener("load", onLoad);
}
```

```
let username = document.querySelector("[name=username]");
username.addEventListener("blur", checkUsername);
```

V tomto řešení jsme použili větší množství funkcí. Pro snazší pochopení se na ně podíváme od konce, protože v takovém pořadí budou volány.

Poslední dva řádky zařídí přidání posluchače události **blur** (seznámili jsme se s ní v minulé podkapitole pro zelenáče), profíci by zde možná použili kontrolu během psaní pomocí události **input**. Staráme se pouze o pole **username**, zbytek HTML formuláře je uveden jen pro úplnost.

Funkce **checkUsername** představuje první část kontroly. Potřebujeme v ní vykonat HTTP požadavek. V klientském JavaScriptu jsou za tímto účelem dostupné dvě funkce: starší **XMLHttpRequest** (často zkracována na **XHR**) a novější **fetch**. My si zatím předvedeme XHR, na **fetch** narazíme později v šesté kapitole. Nenecháme se zastrašit dlouhým názvem s pochybnou velikostí písmen; tento objekt pochází z doby, kdy se po síti data často přenášela ve formátu XML. To dnes tolik neplatí a **XMLHttpRequest** rádi použijeme na přenos dat ve formátu jiném (např. JSON).

Proměnná **xhr** zastřešuje naplánovaný HTTP požadavek. Nejprve mu musíme v souladu s použitým API nastavit adresu. Její součástí je hodnota získaná z formulářového pole (proč? protože musíme serveru sdělit, jaké že jméno se uživatel pokouší zaregistrovat). K sestavení výsledného URL jsme použili dva triky:

1. Funkce **encodeURIComponent** upraví zadaný řetězec tak, aby bylo bezpečné jej použít jako hodnotu v *URL query stringu*, tj. v části webové adresy za otazníkem. Data v tomto prostoru používají speciální syntaxi postavenou mj. na znacích **=** a **&**. Pokud by se takové znaky náhodou (nebo dokonce úmyslně – pamatujeme, že někteří uživatelé jsou potouchlí) nacházely v **username.value**, rozbilo by to strukturu výsledné adresy. Proto tyto problémové znaky nejprve pomocí **encodeURIComponent** převedeme na jejich bezpečnější zápis.
2. Proměnnou **url** definujeme jako řetězec ohraničený dvojicí *zpětných apostrofů* (anglicky *backtick*). Jedná se o méně obvyklý znak, který se na anglické klávesnici nachází nalevo od jedničky. Tyto řetězce mají v JavaScriptu speciální funkcionalitu: pokud se v nich objeví znak dolaru a složené závorky, je tato posloupnost nahrazena hodnotou uvnitř závorek. Jedná se o ideální

způsob, jak kombinovat více řetězců nebo doplňovat proměnné hodnoty na místa v pevně definovaných šablonách. Proto se těmito řetězcům říká *template literals*.

Jakmile máme nachystáno cílové URL, dokončíme konfiguraci proměnné **xhr** v těchto krocích:

- Vlastnost **responseType** říká, v jakém datovém typu očekáváme odpověď od serveru. Výchozí hodnota **"text"** je vhodná pro přenos textových dat; náš backend vrací data strukturovaná pomocí JSON. Pokud budou data skutečně v tomto formátu, XMLHttpRequest je rovnou převede na JavaScriptový slovník.
- Metoda **open** určuje použitou HTTP metodu a cílové URL.
- Metoda **send** slouží především k předání dat, která potřebujeme na server odeslat v těle požadavku. My žádná taková nemáme (pro metodu **GET** ani mít nemůžeme), neboť poptávané uživatelské jméno předáme již v URL.
- Objekt XMLHttpRequest generuje DOM události, takže přidáme posluchač na **load**. To je událost, která vznikne, jakmile server vrátí námi vyžádaná data.

V tomto místě je dobré se na chvíli pozastavit a všimnout si, že vykonání HTTP požadavku je **asynchronní operace**. Takto označujeme funkcionalitu, která je vykonávána *na pozadí*, tedy paralelně s JavaScriptovým kódem, který po ní následuje. Zkušenější programátor by možná očekával, že metoda **xhr.send()** bude *synchronní* (někdy též *blokující*), tj. že prohlížeč během jejího vykonávání provede zmíněný HTTP požadavek a teprve po jeho dokončení se bude pokračovat. Jenže to by mohlo trvat velmi dlouho a došlo by tak k pozastavení našeho kódu, který by kvůli čekání na HTTP požadavek nemohl dělat nic dalšího. Proto je objekt XMLHttpRequest asynchronní. Tato jeho vlastnost dokonce vedla ke vzniku hovorového označení **Ajax**, které bylo kdysi módní používat právě pro tento způsob přenosu dat (Ajax = *Asynchronous JavaScript and XML*).

Následující, resp. předchozí funkce **onLoad** je tedy posluchač události a jako taková dostává parametr s objektem události. To je pro nás dobrý způsob, jak se v ní dostat k proměnné **xhr** (připomeňme, že **e.target** je objekt, který událost

vyvolal). Vlastnost **response** pak odpovídá datům, která poslal server. Pokud se cestou nic nepokazilo, bude to slovník odpovídající vzorovým datům z úvodu kapitoly. O dalším chování pak rozhodneme podle jeho vlastnosti **available**.

Když už máme data ze serveru, chceme na jejich základě uživatele informovat o (ne)dostupnosti uživatelského jména. Za tímto účelem zobrazíme pod formulářovým polem varovnou hlášku, pokud je jméno zabráněno. Musíme si dát ale pozor na situaci, kdy tato kontrola probíhá opakovaně, a tak je možné, že tato hláška už ve formuláři je. Vlastně mohou nastat celkem čtyři situace:

1. Žádnou hlášku nezobrazujeme a podle odpovědi serveru ani nemusíme: v takovém případě neděláme nic.
2. Žádnou hlášku nezobrazujeme, ale podle odpovědi serveru bychom nově měli: bude nutné ji vyrobit a zobrazit.
3. Hláška už je zobrazena, ale uživatel změnil jméno a to nové už je v pořádku: starou hlášku budeme muset skrýt.
4. Hláška už je zobrazena a ze serveru přišla opět instrukce k jejímu ukázání: budeme muset vyrobit novou hlášku s novým textem a tu starou jí nahradit.

Vzorové řešení má pro tyto případy funkce **hideError** a **showError**, které voláme dle získaných instrukcí. Obě jsou připraveny na situaci, že už chybový text zobrazujeme: pokud nějaký naleznou, tak jej nejprve odstraní. K nalezení se hodí poslední pomocná mini-funkce **getError**, v jejíž implementaci se můžeme setkat s další drobnou novinkou z rozhraní DOM. Jedná se o vlastnost **labels**, která je dostupná u všech formulářových prvků a odpovídá poli HTML značek **<label>**, jež jsou s daným prvkem spárovány v roli popisku.

Co jsme se naučili

Po vyřešení čtvrté úlohy by měl čtenář chápat a ovládat:

- koncept komunikace se serverem pomocí HTTP JSON API
- práci s asynchronním objektem **XMLHttpRequest**
- vlastnost **labels** u formulářových polí

Zelenáči: řízení toku kódu pomocí operátorů

V našich funkcích se často rozhoduje o chování programu na základě hodnot uložených v proměnných. Typickým představitelem takového rozhodování je klíčové slovo **if**, resp. konstrukce **if (podmínka) { ... } else { ... }**. Využijme tuto podkapitulu, abychom si ukázali další možnosti, jak realizovat podmínky a větvení kódu.

Budeme k tomu potřebovat *operátory*. Jsou to syntaktické prvky, zpravidla tvořené jedním či dvěma speciálními znaky, které slouží především k provedení nějaké operace s jednou či více veličinami. Dobrým příkladem je operátor **+** (sčítání či řetězení), se kterým se setkáváme od základní školy. Operátory dělíme do skupin podle toho, s kolika hodnotami (říkáme jim *operandy*) pracují. Unární operátory (např. **!**) mají jen jeden operand, binární dva, ternární operátor potřebuje operandy tři.

A právě ternární operátor (tvoří jej znak **?** v kombinaci se znakem **:**) nás nyní zajímá. Jeho tradiční použití vypadá takto:

```
let mince = Math.random();  
let text = (mince < 0.5 ? "panna" : "orel");
```

Funkce **Math.random()** vrátí náhodnou hodnotu mezi nulou a jedničkou. Ternární operátor na druhém řádku jsme pro zvýšení čitelnosti vložili do kulatých závorek, ale není to nutné. Jeho první operand – výraz před otazníkem – je testovaná podmínka. Pokud platí, ternární operátor vrátí svůj druhý operand, zapsaný mezi otazníkem a dvojtečkou. V opačném případě vrátí třetí operand (uvedený za dvojtečkou).

A zde je vlastnost, která se nám může hodit: jako druhý a třetí operand nemusíme použít jen prostou hodnotu, ale libovolně složitý JavaScriptový výraz. Hodnota celého ternárního operátoru pak bude podle podmínky vybrána jako hodnota prvního či druhého výrazu. Pojďme zkusit použít ternární operátor jako alternativu **if** v rámci funkce **onLoad**. Její stávající tvar je tento:

```
function onLoad(e) {  
  let data = e.target.response;  
  if (data.available) {  
    hideError();  
  }  
}
```

```

    } else {
        showError(data.suggested);
    }
}

```

Použitím ternárního operátoru se kód zkrátí:

```

function onLoad(e) {
    let data = e.target.response;
    data.available ? hideError() : showError(data.suggested);
}

```

Za zmínku stojí, že návratová hodnota ternárního operátoru je v tuto chvíli nepodstatná. Abychom ještě více rozšířili svůj programátorský arzenál, prozkoumáme nyní dva další – tentokrát binární – operátory. Jedná se o **&&** (zvaný *a zároveň*) a **||** (zvaný *nebo*).

Programátoři všech úrovní znalostí se s těmito operátory setkávají zejména při nutnosti zkombinovat více podmínek; často je potkáváme v situacích podobných těmto příkladům:

```

if (usernameIsOk && passwordIsOk) { /* dokončit registraci */ }

if (amount == 0 || amount > 10) { /* toto množství nelze objednat */ }

```

Tato forma použití v nás může budit dojem, že výsledkem těchto operátorů je pravdivostní hodnota **true/false** (tzv. *boolean*). To je ovšem značně nepřesné. Lepší definice zní takto:

- Výsledkem operátoru **&&** je hodnota prvního operandu, pokud je *pravdivá*. V opačném případě je výsledkem hodnota druhého operandu.
- Výsledkem operátoru **||** je hodnota prvního operandu, pokud je *nepravdivá*. V opačném případě je výsledkem hodnota druhého operandu.

(Termíny *pravdivá* resp. *nepravdivá* odpovídají intuitivnímu procesu převodu libovolné hodnoty na boolean. Tušíme, že **false** bude nula, prázdný řetězec a podobně. Vše ostatní je **true**. Přesněji se na tyto převody podíváme hned v následující podkapitole pro koumáky.)

Tyto operátory a jejich chování můžeme použít k řízení toku kódu díky další jejich vlastnosti, která se nazývá **zkrácené vyhodnocování** (anglicky *short-circuit evaluation*). Ta říká, že pokud o hodnotě operátoru lze rozhodnout na základě prvního operandu, ten druhý bude ignorován. Pomocí této úvahy můžeme nyní lehce zjednodušit kód ve funkci `hideError`. Její původní implementace vypadala takto:

```
function hideError() {
  let error = getError();
  if (error) { error.remove(); }
}
```

Využitím zkráceného vyhodnocování můžeme tento kód upravit:

```
function hideError() {
  let error = getError();
  error && error.remove();
}
```

Stejně jako v minulém příkladu, i nyní nás nezajímá výsledek operátoru. Využíváme jen jeho vlastnosti, že druhý operand (metoda `error.remove()`) bude vykonán jen pokud je první operand *pravdivý*.

Pro operátor `||` teď žádné vhodné využití nemáme, ale dostaneme se k němu ještě v kapitole 12.

Koumáci: *falsy values* a operátory s implicitním přetypováním

Z předchozí podkapitoly jsme si odnesli nové informace o některých operátorech, ale také koncept toho, že libovolnou hodnotu v JavaScriptu lze – většinou pro potřeby podmínek – převést na pravdivostní hodnotu. Děje se tak buď implicitně (např. při zápisu `if (hodnota) { ... }`), nebo explicitně použitím *unárního operátoru negace* zapsaného znakem vykřičníku (tento operátor svůj operand nejprve převede na boolean a pak vrátí opak). Pro dobré porozumění hodnotám v JavaScriptu musíme vědět, jak tento převod probíhá.

Existuje poměrně malá množina hodnot, které jsou převedeny na **false**. Říkáme jim *falsy values* a jejich seznam je následující:

- pravdivostní hodnota **false**
- číslo **0** a speciální číselná hodnota **NaN** (*Not a Number*)
- prázdný řetězec
- speciální hodnoty **null** a **undefined**

(Mimochodem, čím že se to vlastně liší **null** a **undefined**? Hodnotu **null** je doporučeno používat explicitně tam, kde chceme jasně vyjádřit prázdko či neexistenci. Hodnota **undefined** pak odpovídá těm případům, kdy potřebná veličina chybí implicitně, neboť ji nikdo nedodal. U proměnných bez přiřazené hodnoty, u přístupu k neexistujícímu klíči ve slovníku, při použití parametru funkce, která jej nedostala předaný při volání.)

Všechny ostatní hodnoty jsou *truthy values* a při převodu na boolean z nich vznikne **true**.

Výše uvedený seznam je dostatečně krátký na to, abychom si jej zapamatovali a dopředu dovedli odhadnout, jak dopadne implicitní přetypování v rámci podmínky nebo zkráceného vyhodnocení operátorů **&&** a **||**. JavaScript bohužel provádí implicitní přetypování také v dalších situacích, které jsou i pro zkušeného programátora výrazně méně předvídatelné. Zde máme na mysli zejména operátor sčítání (znak **+**) a operátor porovnání (znaky **==**). Pojďme si jejich specifika prohlédnout na příkladech.

U sčítání narážíme na skutečnost, že tento operátor plní dvě různé role: aritmetickou operaci pro čísla a spojení (zřetězení) pro řetězce. Tato flexibilita s sebou nese daň v podobě komplexní logiky, kterou operátor sčítání provádí v situaci, kdy jeho operandy nespádají do jedné z těchto dvou kategorií:

```
1 + "1";      // "11"
true + true;  // 2
[] + [];      // ""
{} + {};      // "[object Object][object Object]"
```

Pokud je čtenář některými výsledky překvapen, pak nabízíme snadný recept, jak se jich vyvarovat: používat operátor sčítání pouze tam, kde jsme si jisti tím, že sčítáme sčitatelné (tj. operandy jsou buď dvě čísla, nebo dva řetězce). O něco komplikovanější je situace s operátorem porovnání, u kterého nám v řadě situací může dávat smysl porovnávání hodnot různých typů. Jsme kupříkladu rádi, že číslo můžeme porovnávat s řetězcem, který obsahuje tu samou hodnotu:

```
"42" == 42; // true
```

V praxi se s podobným porovnáním můžeme setkat často; třeba když bereme číselnou hodnotu z formulářového pole (neboť `input.value` je řetězec). Ale takové pohodlí s sebou bohužel přináší nepříjemné výsledky:

```
"0" == 0; // true
0 == ""; // true
"" == "0"; // false
```

Dávají vám jednotlivá porovnání smysl? Sama o sobě jsou logická, ale dohromady zde máme tři hodnoty: první dvě se rovnají, druhé dvě se rovnají a první se třetí nikoliv. Nalezli jsme příklad, kdy **operátor rovnosti není tranzitivní**. Aby se s ním pracovalo komfortně, musí obsahovat komplikovanou sadu pravidel pro situace, kdy jeho operandy mají různý typ. A tato pravidla si málokdo dokáže zapamatovat, což snižuje předvídatelnost operátoru.

Pokud nás tato flexibilita trápí, můžeme sáhnout po jednom ze dvou řešení:

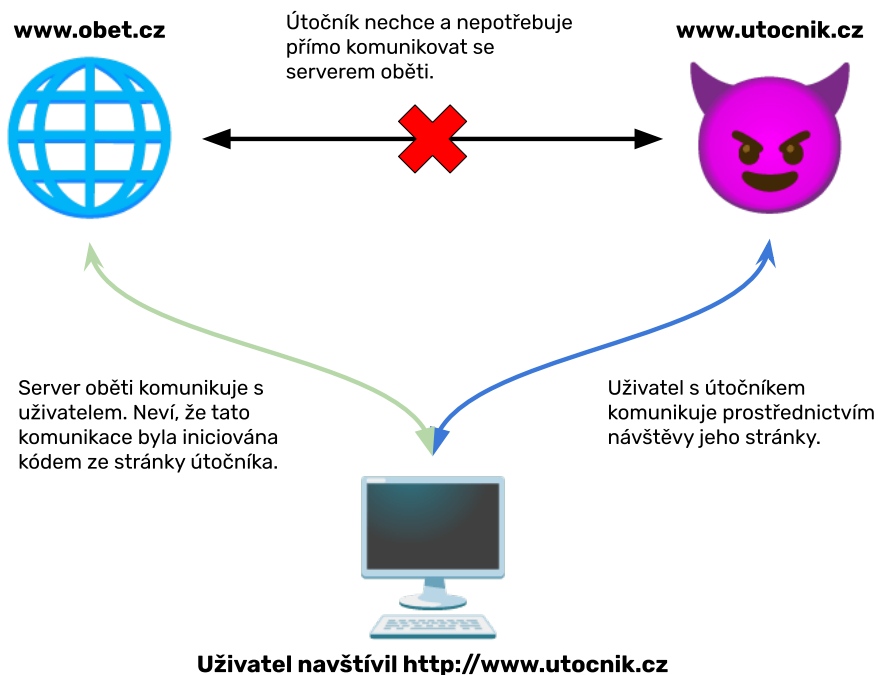
1. Namísto operátoru rovnosti (`==`) používat trojrovnítkový **operátor ekvivalence** (`===`). Ten funguje obdobně jako běžné porovnání, ale pokud jeho operandy mají různý typ, ihned vrátí **false**. Neprojde skrz něj tedy žádný ze tří testů uvedených v předchozí ukázce.
2. Používat porovnávání jen mezi proměnnými shodných datových typů, resp. těch typů, u kterých jsme si jisti porovnávacím algoritmem. Je to stejné doporučení, jako to před chvílí zmíněné u operátoru sčítání.

Profíci: *same-origin policy* a CORS

Pojďme si nyní od JavaScriptu trochu odpočinout a podívat se na téma lehce odlišné. Ve vzorovém řešení jsme viděli, že v rámci webové stránky lze vytvořit HTTP požadavek *na pozadí*, a dostat se tak k dalším datům, případně data přenést na server, aniž bychom provedli *navigaci* (tj. uživatele vzali na jinou stránku). Jde o velice silnou techniku, a tak je zde otázka, jestli tím klientskému JavaScriptu náhodou nedáváme do ruky nástroje, které by bylo možné zneužít.

Opatrnost je na místě. Webové stránky jsou, v kontrastu s běžným programovým vybavením počítače, ve specifickém postavení. Uživatel je často navštíví, aniž by dopředu věděl, co ho čeká. Prostřednictvím prohlížeče vidí web, který mohl vytvořit autor nekalých úmyslů (v tomto kontextu se často celkem přímočaře rovnou říká *útočník*). Automaticky nelze rozhodnout o tom, zdali je účelem stránky uživateli uškodit, takže prohlížeč namísto toho poskytuje ochranu pasivní: nedovolí žádnou operaci, která by byla zneužitelná. Proto například v klientském JavaScriptu nemůžeme přistupovat k souborům uživatele.

A jak je to tedy s HTTP požadavky? Je jasné, že útočník disponující vlastním webovým serverem může vykonávat libovolnou síťovou komunikaci iniciovanou *právě z toho jeho serveru*. Když ale svůj kód vyrobí v JavaScriptu, dojde k jeho vykonání v prohlížeči uživatele; požadavky budou tedy vycházet z *uživatelova počítače*. Zhruba tak, jako na obrázku:



Obrázek: Útočník může komunikovat s obětí prostřednictvím nic netušícího uživatele

Takové nastavení otevírá útočníkovi možnost vykonávat požadavky, které by jinak nemohl. Konkrétně:

1. Požadavky na servery, které jsou ochotny provést komunikaci s počítačem uživatele, ale nikoliv se serverem útočníka.
2. Požadavky, do kterých uživatelův prohlížeč automaticky doplňuje nějaká *tajemství*, kterými útočník nedisponuje. Může jít o HTTP cookies, jméno a heslo pro HTTP autorizaci, klientské certifikáty a podobně. V roce 2024 sice v rámci omezování tzv. *3rd party cookies* ztrácí tento útok na významu, nicméně předchozí bod stále platí v plné šíři.

V praxi proto prohlížeče silně omezují možnosti HTTP požadavků (jak **XHR**, tak **fetch**). Tato ochrana se nazývá *Same Origin Policy* (SOP) a ve skutečnosti je velice jednoduchá. K jejímu pochopení stačí definovat termín **origin webové adresy** – jedná se o tři její komponenty, konkrétně *schema*, *hostname* a *port*. Tyto tvoří první část URL až po první lomítko za doménou. Nejlépe to uvidíme na příkladu: vzorová

adresa <https://karel.cz/pisnicky/texty.html> má origin <https://karel.cz>. SOP pak prostě předepisuje, že JavaScriptové požadavky smíme provádět jen pokud se **origin cílového URL shoduje s originem stránky, na které se nacházíme**.

V řadě případů se nás SOP nijak nedotkne. Pokud je cílová adresa na serveru, jehož stránku zobrazujeme, omezení se našeho kódu netýká. To zejména zahrnuje všechny relativní adresy začínající lomítkem (častá praxe), neboť ty z definice vedou na stejný origin, jako má stránka. Kdybychom ale chtěli načítat data z úplně jiného serveru, budeme muset situaci se SOP nějak vyřešit.

Pro zdárný přenos takovýchto dat máme jen dvě smysluplné možnosti:

- Situace, kdy chceme provést HTTP požadavek, ale nepotřebujeme následně pomocí JavaScriptu přistupovat k jeho odpovědi. Je to nepravděpodobný, ale možný scénář. Zahrnuje například zobrazení obrázků z jiné domény, vložení stylu či skriptu a podobně. Taková aktivita není útočnickem zneužitelná, proto ji SOP nezahrnuje – tato ochrana se projeví teprve ve chvíli, kdy bychom se zajímali o data získané odpovědi.
- Situace, kdy se s provozovatelem cílového originu domluvíme na tom, že mu takový druh komunikace nevadí. Samozřejmě zde nemáme na mysli skutečnou ústní dohodu; stačí, aby vzdálený webový server nějakým mechanismem **povolil, že mu nevadí zpracovat požadavek pocházející z jiného originu**.

Druhá popisovaná situace je velmi častá. Zmiňovaný souhlas je v prohlížeči realizován technikou nazvanou CORS (*Cross-Origin Resource Sharing*). Díky ní je možné za určitých podmínek tvrdá omezení SOP obejít. Celá rozhodovací posloupnost pak vypadá následovně:

1. Uživatel se nachází na stránce s originem A, její JavaScript se pokouší provést požadavek na URL s originem B. Pokud se tyto originy shodují, je vše povoleno a není co řešit.
2. Prohlížeč provede požadavek, ale přidá k němu v HTTP hlavičce **Origin** zmínku o tom, že autor požadavku – klientský počítač – zobrazuje stránku na originu A.

3. Od serveru z originu **B** přijde odpověď. Prohlížeč se podívá, jestli se v jejích hlavičkách nachází informace o tom, že server souhlasí (tzv. CORS hlavička). Pokud ne, odpověď se před JavaScriptem zatají (SOP). Pokud ano, odpověď se JavaScriptovému kódu předá.

Toto schéma jsme záměrně trochu zjednodušili. Zvědavý čtenář se o pokročilejších konceptech CORS (další hlavičky, technika *preflight*) může dočíst například na MDN (<https://developer.mozilla.org/>), nebo na webu enable-cors.org.

Cesta k SPA, riziko XSS

Úloha

Na našem webu, který fanouškům poskytuje texty písní Karla Gotta, nabízíme funkci **hledání**: uživatel zadá do formulářového pole hledaný text a my mu po odeslání na nové stránce zobrazíme všechny související písně. Rádi bychom současné řešení upravili na modernější SPA (*single-page application*), kdy při zobrazení výsledků nedochází k načtení nové stránky. Backend bude data opět nabízet formou HTTP JSON API.

Řešení

Stejně jako v minulé kapitole, i zde máme v plánu pracovat s daty, která získáme JavaScriptovým požadavkem z backendu. Komunikace může vypadat třeba takto:

- data odesílaná na server budou realizována HTTP požadavkem poslaným na adresu `/search?query=...`
- server odpoví daty ve formátu JSON a bude to pole; jeho jednotlivé položky budou vypadat takto:

```
{
  "title": "Mám styl Čendy",
  "text": "Mezi námi je <em>mnoho</em> chvil",
  "url": "..." // odkaz na celý text písně
}
```

Pod klíčem **text** je uložena část textu písně, díky které se píseň dostala do výsledků. Konkrétně server značkou `` označí tu část textu, která odpovídá hledanému termínu.

Mimochodem: je nezbytné, aby to dělal server, když i klient zná hledaný termín a mohl by jej v textu najít a označit? Ano, v naprosté většině případů je zodpovědností serveru, aby data takto připravil. Jen on totiž ví, proč se píseň dostala do výsledků; možná že je v ní hledaný termín v jiném pádu, bez diakritiky nebo s jinou velikostí písmen.

Vzorové řešení pak může vypadat takto:

```
<!-- kapitola-6.html -->
<h1>Hledání</h1>
<form>
  <label>
    Hledaný výraz: <input type="text" name="query" />
  </label>
  <button>🔍</button>
</form>

<section id="results"></section>

<script src="kapitola-6.js"></script>
```

```
// kapitola-6.js
let form = document.querySelector("form");
let results = document.querySelector("#results");

function buildSong(song) {
  let item = document.createElement("li");
  item.innerHTML = `
    <a href="${song.url}">${song.title}</a>
    <br/> ${song.text}
  `;
  return item;
}

function showResults(xhr, query) {
  let songs = xhr.response;
  if (songs.length == 0) {
    results.replaceChildren("Dotazu nevyhovují žádné písně 😞");
  }
}
```

```

    return;
}

let heading = document.createElement("h2");
heading.textContent = `Nalezené písně pro dotaz: ${query}`;

let ol = document.createElement("ol");
results.replaceChildren(heading, ol);

for (let i=0; i<songs.length; i++) {
    let song = buildSong(songs[i]);
    ol.append(song);
}
}

function onSubmit(e) {
    e.preventDefault();
    let xhr = new XMLHttpRequest();
    let query = form.querySelector("[name=query]").value;
    let url = `/search?query=${encodeURIComponent(query)}`;
    xhr.responseType = "json";
    xhr.open("GET", url);
    xhr.send();
    xhr.addEventListener("load", e => showResults(xhr, query));
}

form.addEventListener("submit", onSubmit);

```

Kód neobsahuje žádné velké novinky. Jedná se o kombinaci minulých dvou kapitol (použití `XMLHttpRequest` a události `submit`). Za zmínku stojí:

- Ve formuláři jsme použili HTML prvek `<button>`, který ve výchozím nastavení též funguje jako odesílací tlačítko.
- V rámci posluchače `onSubmit` vždy voláme `e.preventDefault()`, abychom tak zamezili odeslání. V kontextu této úlohy to neznamená chybu uživatele, ale snahu zůstat na stejné stránce a požadovanou funkcionalitu následně vykonat pomocí JavaScriptu.

- Data získaná z hledacího pole před vložením do URL opět upravujeme funkcí `encodeURIComponent` pro případ, že by obsahovala znaky, které do URL nepatří.
- Posluchač události `load` je malá anonymní arrow funkce, která uzavírá proměnné `xhr` a `query`. Díky tomu můžeme do `showResults` předat libovolné parametry.

V HTML dokumentu jsme si nachystali prázdný prvek `<section id="results">`, jehož obsah následně plníme na základě dat ze serveru. To je velmi častý postup: v HTML připravíme jen kostru či šablonu výsledné stránky, JavaScriptem do ní později dodáme potřebné informace. V těchto případech je dobré nezapomenout na dva scénáře:

1. Co uživatel vidí, dokud se čeká na odpověď od serveru? Jestli to je nějaká neúplná HTML struktura, měla by být skryta. V našem případě nevidí nic.
2. Co uživatel vidí, pokud se prázdný prostor ve stránce plní opakovaně? Při každém hledání je nutné obsah prvku `#results` nahradit. Namísto přidávání nových uzlů proto používáme `results.replaceChildren()`.

V této kapitole se pojďme podívat pozorněji na funkci `buildSong`, která slouží k výrobě HTML prvku odpovídajícímu jednomu výsledku hledání. Poprvé se v ní setkáváme s vlastností `innerHTML`, která patří mezi nejsilnější součásti rozhraní DOM. Když nějakému HTML prvku přiřazujeme do vlastnosti `innerHTML`, říkáme tím, že prohlížeč má celý obsah (potomky) tohoto prvku nahradit novým podstromem, který vznikne parsováním zadaného řetězce. Je to skoro stejné, jako když necháváme prohlížeč sestavit strom dokumentu při prvním načtení stránky.

Jedná se o velmi pohodlný způsob tvorby složitější komponenty webové stránky. Porovnejme, o kolik by bylo zdoluhavější výsledek hledání vyrobit a sestavit pomocí několika volání `createElement` a `append`. Zároveň je pro nás řešení pomocí `innerHTML` nezbytné, protože od serveru již dostáváme malé části HTML – vlastnost `text` s úryvkem textu písně obsahuje značku pro zvýraznění.

I přes nezpochybnitelné pohodlí vlastnosti `innerHTML` si ale musíme dát velký pozor na její použití. Jedná se totiž bohužel o častý vstupní bod pro zranitelnost typu **XSS** (*cross-site scripting*). Abychom lépe pochopili, oč jde, můžeme se podívat hned o pár řádků vedle, do funkce `showResults`. Tam se vyrábí nadpis pro výsledky:

```
let heading = document.createElement("h2");
heading.textContent = `Nalezené písně pro dotaz: ${query}`;
```

Tentokrát se namísto `innerHTML` používá `textContent`, který taktéž nahradí danému uzlu obsah, ale prostým textem. Zadaný řetězec není parsován jako HTML, tj. případné HTML značky v něm obsažené se zobrazí jen jako text. A je to tak správně, neboť při použití `innerHTML` by pak mohl záludný uživatel do hledacího pole napsat například:

```

```

Náš kód by pak v nadpisu namísto zadaného řetězce ukázal obrázek. A to je velká chyba, neboť uživatel svým vstupem dokázal **pozměnit strukturu dokumentu**. Od toho je pak už jen krůček k tomu, aby tímto způsobem mohl vložit vlastní skript. Jak přesně by toho docílil a jaké nebezpečí by z toho mohlo plynout – to je mimo rozsah této knihy. Nám bohatě stačí, že bychom tak nechali uživatele do námi připraveného dokumentu vkládat jeho značky. To je v naprosté většině případů základ pro bezpečnostní malér.

Bereme si z toho velmi důležité ponaučení: jakmile vytváříme HTML dokument nebo jeho část, musíme uvažovat, jaká data v něm zobrazíme. Pokud existuje možnost, že tato data nepochází z důvěryhodného zdroje (tj. mohou obsahovat neočekávané HTML znaky), pracujeme defenzivně a tato data vkládáme pomocí `textContent` (nebo jako parametry do `append()`). Jen tak je zaručeno, že prohlížeč zadaný řetězec nebude považovat za HTML. A teprve když jsme si absolutně jisti, že v zobrazovaných proměnných jsou jen ne-HTML data, smíme použít mocnou vlastnost `innerHTML`.

Pro úplnost: je výše uvedenou optikou bezpečné naše použití `innerHTML` ve funkci `buildSong`? Záleží na tom, odkud se berou data uložená pod klíči `url`, `text` a `title`. Pokud bychom uvažovali variantu, že by Karel Gott do nějakého svého textu umístil záludný kousek HTML, museli bychom od serveru vyžadovat, aby vrácená data náležitě zabezpečil (problémové znaky nahradil za HTML entity).

Co jsme se naučili

Po vyřešení páté úlohy by měl čtenář chápat a ovládat:

- podstatu zranitelností XSS
- rozdíl mezi vlastnostmi `textContent` a `innerHTML`

Zelenáči: práce s adresním řádkem

SPA, tedy jednostránkové webové aplikace, staví na technice *nahrazování obsahu JavaScriptem* namísto tradičního mechanismu *navigace* mezi více různými stránkami. V porovnání s běžně odesílaným formulářem je ale naše současné řešení ještě pořád trochu nemotorné. Při hledání totiž nedochází ke změně URL v adresním řádku prohlížeče. A to je škoda, protože tak přicházíme o:

- možnost uložení adresy s výsledky do záložek,
- možnost poslání takové adresy někomu jinému,
- možnost reloadu (opětovného načtení) stránky s výsledky,
- pohyb v historii prohlížeče (tj. tlačítko *Zpět*).

Uživatel se stále nachází na té samé stránce, takže v adresním řádku je stále jen např. **search.html**. Zobrazovaná data se ovšem mění – dávalo by tedy smysl, aby se měnilo i zobrazované URL.

Řešení není komplikované, byť na něj vývojáři často a rádi zapomínají. Sestává z těchto kroků:

1. Po provedení hledání musíme informaci o hledaném termínu vložit do URL v adresním řádku. To mj. způsobí záznam do historie prohlížeče, takže pak bude možný návrat tlačítkem *Zpět*.
2. Jakmile uživatel toto tlačítko použije, URL se změní (na předchozí hodnotu), ale my zůstáváme na stejné stránce. Musíme proto zobrazit obsah korespondující s novým URL.
3. Pokud uživatel načte novou stránku, v jejímž URL bude námi poznamenaná hledaná hodnota, musíme provést hledání.

Pojďme část vzorového řešení v tomto smyslu upravit a rozšířit. Nejprve rozdělíme *odeslání formuláře* a *hledání* do dvou funkcí:

```
function search(query) {  
  let xhr = new XMLHttpRequest();  
  let url = `/search?query=${encodeURIComponent(query)}`;  
  xhr.responseType = "json";  
  xhr.open("GET", url);  
  xhr.send();  
  xhr.addEventListener("load", e => showResults(xhr, query));  
}  
  
function onSubmit(e) {  
  e.preventDefault();  
  let query = form.querySelector("[name=query]").value;  
  search(query);  
}
```

To odpovídá plánu, že hledání bude vyvoláno i jinými mechanismy, než jen odesláním formuláře. Dále, po odeslání formuláře bude potřeba změnit URL v adresním řádku:

```
function onSubmit(e) {  
  e.preventDefault();  
  let query = form.querySelector("[name=query]").value;  
  
  let url = new URL(location.href);  
  url.searchParams.set("query", query);  
  history.pushState("", "", url);  
  
  search(query);  
}
```


Zde vidíme dvě novinky:

1. Objekt **URL** nabízí pohodlnou práci s webovými adresami. V našem případě je nejzajímavější pod-objekt **searchParams**, pomocí kterého můžeme snadno přistupovat k části URL za otazníkem. Zpravidla se jí říká *search parameters* a tradičně do ní vkládáme řetězce ve tvaru klíč-hodnota. Jejich konkrétní formát nemusíme řešit, neboť to za nás zařídí právě objekt **URL**.
2. Proměnnou **history**, která nám mj. dovoluje měnit hodnotu v adresním řádku bez nutnosti navigace na nový dokument. V uvedeném řešení k tomu používáme metodu **pushState**.

Tím jsme vyřešili první krok ze tří. Pro ten druhý se musíme dozvědět o tom, že došlo k uživatelem vyvolané změně adresního řádku. K tomu slouží událost **popstate**, která nastává na globálním objektu **window**:

```
function load() {
  let url = new URL(location.href);
  let query = url.searchParams.get("query");
  query && search(query);
}
window.addEventListener("popstate", e => load());
```

Jedná se o proces symetrický k odeslání formuláře. Prohlédneme aktuální adresu, a pokud je v ní zaznamenán hledaný text, provedeme hledání.

Poslední třetí krok je jen pomyslná třešnička na dortu, protože k jeho splnění stačí novou funkci **load** zavolat po prvním načtení stránky. Celý upravený kód proto bude vypadat takto:

```
function search(query) {
  let xhr = new XMLHttpRequest();
  let url = `/search?query=${encodeURIComponent(query)}`;
  xhr.responseType = "json";
  xhr.open("GET", url);
  xhr.send();
  xhr.addEventListener("load", e => showResults(xhr, query));
}
```

```

function onSubmit(e) {
  e.preventDefault();
  let query = form.querySelector("[name=query]").value;

  let url = new URL(location.href);
  url.searchParams.set("query", query);
  history.pushState("", "", url);

  search(query);
}

function load() {
  let url = new URL(location.href);
  let query = url.searchParams.get("query");
  query && search(query);
}

form.addEventListener("submit", onSubmit);
window.addEventListener("popstate", e => load());

load();

```

Pozorného čtenáře možná napadla otázka, kdy je správná chvíle na propsání právě hledaného termínu do URL. Jistým pohledem by dávalo smysl tuto akci vykonat uvnitř funkce **search**, protože právě při hledání má dojít ke změně v adresním řádku. Nabídneme dva argumenty, proč je naše současné řešení vhodnější:

1. Funkce **search** zůstává určená pouze k provedení hledání. Pokud bychom do ní vložili i změnu URL, došlo by k přílišnému rozšíření její zodpovědnosti (byla by *moc chytrá*). Znamenalo by to například, že není možné provést hledání bez změny URL.
2. Funkci **search** voláme i v situacích, kdy změna URL není nutná, resp. žádoucí. Je to jednak při prvním načtení stránky (v důsledku volání funkce **load**) a jednak při změně v adresním řádku (v důsledku události **popstate**). V obou těchto případech už v URL správná data jsou.

Výše uvedené pozorování můžeme shrnout do poučky, která platí téměř ve všech podobných situacích: **Změna hodnoty v adresním řádku by měla být vyvolána jen v důsledku uživatelské interakce.**

Koumáci: funkcionální iterace

Ve třetí kapitole jsme si představili alternativní možnosti iterace polí. Pojďme si nyní na úloze z této kapitoly vyzkoušet refactoring kódu pomocí funkcionální iterace.

Podstatou funkcionální iterace je opakované používání malých funkcí vykonávaných automaticky nad položkami v poli. Ideálním startovním bodem je funkce **showResults**, ve které se vytváří jednotlivé výsledky hledání. Zajímá nás tato její část:

```
for (let i=0; i<songs.length; i++) {
  let song = buildSong(songs[i]);
  ol.append(song);
}
```

Procházíme pole **songs** a pro každou jeho položku vyrobíme HTML prvek. To je úloha pro funkcionálně-iterační metodu **map**. Její použití nad polem vrátí nové pole, jehož každá položka vznikla vykonáním malé iterační funkce nad položkou pole původního. V našem případě by to mohlo vypadat takto:

```
let items = songs.map(buildSong);
ol.append(...items);
```

Funkce **buildSong** dostává jako (první) parametr jednotlivé položky odpovědi a vrací nově vzniklý HTML prvek ****. V proměnné **items** je tedy pole HTML prvků. Ty bychom rádi naráz vložili do seznamu **ol**, ovšem metoda **append** neumí pracovat s polem. Je nicméně variadická, tj. umí přijmout libovolný počet parametrů. Použijeme proto operátor tří teček **...** (nazývá se *spread operator*), jehož úkolem je převést hodnoty pole na jednotlivé parametry funkce. Jedná se o jakési *rozbalení* položek pole tam, kde jsou očekávány položky oddělené čárkou.

Použitím funkce **map** se kód nejen zkrátil a zpřehlednil, ale je i výkonnější: nemusíme volat metodu **append** tolikrát, kolik vypisujeme výsledků.

Zatím jsme si ukázali dvě metody užitečné pro funkcionální iteraci, **forEach** a **map**. Je to jen drobná ochutnávka z širokého množství metod, které nám JavaScriptová pole nabízí. Pojďme si ještě ukázat jednu další, která se často hodí: **filter**. Jejím parametrem je opět malá funkce (někdy se jí říká *predikát*), která bude vykonána nad každou položkou pole. Úkolem této funkce je vrátit pravdivostní hodnotu **true** či **false**. Výsledkem volání **filter** je potom nové pole, které obsahuje jen ty položky původního pole, pro které byla predikátem vrácena hodnota **true**.

Představme si, že backend v rámci nalezených výsledků vrátí i rok, ve kterém píseň vznikla. Použije k tomu nový klíč **year**, jehož hodnotou je číslo. Jedna položka odpovědi tedy vypadá takto:

```
{
  "title": "Mám styl Čendy",
  "text": "Mezi námi je <em>mnoho</em> chvil",
  "url": "https://example.com/",
  "year": 1984
}
```

Kdybychom chtěli vypsát jen ty *nové* písně, které vznikly v roce 1984 a později, napsali bychom si nejprve malý testovací predikát:

```
function isNew(song) {
  return (song.year >= 1984);
}
```

Ve výpisu bychom pak použili navíc metodu **filter**:

```
let items = songs.filter(isNew).map(buildSong);
ol.append(...items);
```

Takové řetězové volání je ve světě funkcionálního programování docela běžné. Zároveň je to místo, kde můžeme s výhodou použít zkráceného zápisu anonymních arrow funkcí:

```
let items = songs.filter(song => song.year >= 1984).map(buildSong);
ol.append(...items);
```

Nyní už samostatný predikát **isNew** nepotřebujeme. Koumáci si po přečtení této podkapitoly možná půjdou dohledat, které další metody pro funkcionální iteraci existují. Bez detailnějších ukázek můžeme napovědět, že ty nejdůležitější ještě neprobrané jsou:

- **reduce** sloužící k vytvoření jediného výsledku na základě všech položek pole (např. součet, průměr, největší hodnota...);
- **some** a **every**, které ověřují, zda některá či všechny položky pole splňují daný predikát;
- **find**, která vrátí první položku pole splňující nějakou podmínku.

U funkcionální iterace si ale ukážeme ještě jednu věc: parametry, které jsou iteračním funkcím předávány. Je zřejmé, že první a hlavní parametr je vždy ta položka pole, kterou právě zpracováváme. Další parametry bychom mohli sami předávat použitím uzávěry. Abychom si práci ušetřili, většina iteračních metod automaticky předává i další dva často užitečné parametry: index (pořadí, od nuly) položky a celé pole, které právě zpracováváme. Kdybychom nepoužívali pro výpis nalezených výsledků číslovaný seznam (HTML značka ****), mohli bychom snadno ve funkci **buildSong** doplnit k názvu i pořadí a celkový počet:

```
function buildSong(song, index, allSongs) {
  let item = document.createElement("li");
  let number = `${index+1}/${allSongs.length}`;
  item.innerHTML = `
    ${number}: <a href="${song.url}">${song.title}</a>
    <br/> ${song.text}
  `;
  return item;
}

let items = songs.map(buildSong);
```

Profíci: fetch, Promises a async/await

V minulé kapitole jsme představili objekt `XMLHttpRequest` a zároveň si slíbili modernější alternativu, totiž funkci `fetch`. Její rolí je taktéž provedení HTTP požadavku, takže hned do začátku se nabízí otázka, v čem že je vůbec použití `fetch` lepší. Fakticky totiž žádnou funkcionalitu, kterou bychom v `XMLHttpRequest` neměli, nenabízí. Hlavní rozdíl tak není v tom, **co** pomocí `fetch` vykonáme, ale **jak**.

Abychom systém práce s funkcí `fetch` dobře pochopili a docenili, budeme se muset nejprve chvíli zabývat **asynchronními** funkcemi v JavaScriptu. To jsou takové, které jako jeden ze svých parametrů přijímají *další* funkci proto, aby ji dříve či později samy vykonaly. Takovému parametru se zpravidla říká **callback** a v této knize jsme zatím potkali dvě místa, kde se callbacky používají:

- metodu `addEventListener`, které dáváme callback k vykonání vždy, když nastane požadovaná událost;
- funkci `setTimeout`, které dáváme callback k vykonání po uplynutí daného času.

Asynchronních funkcí je mnohem více a typicky se s nimi setkáváme u aktivit, které trvají – vágně řečeno – **dlouho**. Aby se prohlížeč při jejich vykonávání nezasekl, necháme je vykonávat potřebnou funkcionalitu *na pozadí* a pomocí callbacku řekneme, co se má stát, až tato dlouhotrvající aktivita skončí. Nutnosti předávání callbacku se říká CPS (anglicky *continuation passing style*). Pro nováčky ve světě JavaScriptu může být takový koncept matoucí – zejména pokud přicházejí z jazyků, ve kterých se asynchronní funkce nevyskytují.

Situaci s CPS dále komplikuje skutečnost, že callback je nutné zkombinovat také s ostatními parametry, které funkce ke svému chování potřebuje. Například u funkce `setTimeout` máme parametry dva: jeden callback a jednu časovou hodnotu. Jejich pořadí si po letech práce stále pletou i velmi zkušení programátoři. Dříve nebo později si začnou všimnout, že u volání asynchronních funkcí se callback tak nějak *nehodí*; že jeho přítomnost je v kódu rušivá a snižuje čitelnost. Proto se v JavaScriptu mezi lety 2012 až 2015 objevil alternativní způsob práce s CPS, zvaný **Promise** (česky *příslib*). Použití Promise nepřináší do jazyka nic koncepčně nového, jedná se jen o tzv. *návrhový vzor*, tedy doporučený způsob, jak řešit často se opakující úlohu.

Při použití Promise se callback nepředává přímo do asynchronní funkce. Namísto toho nám asynchronní funkce vrátí speciální hodnotu (nazvanou Promise), která vyjadřuje skutečnost, že přestože funkce již skončila, její práce ještě není hotová. Callback pak předáme k takto získané Promise její metodou **then**. Můžeme si to prohlédnout na hypotetickém příkladu modernější varianty funkce **setTimeout**:

```
function done() {  
  console.log("hotovo!");  
}  
  
// starý způsob  
setTimeout(done, 500);  
  
// nový způsob  
let promise = setTimeout2(500);  
promise.then(done);
```

Tato ukázka je jen teoretická, protože **setTimeout2** neexistuje – ale kdyby ji dnes někdo navrhl, jistě by fungovala takto. Na první pohled to nevypadá, že bychom pomocí Promise získali nějaký užitek. Jakmile však náš kód začne být složitější, ukáže se, že práce s Promises jej výrazně zjednodušuje.

Pro lepší pochopení můžeme na objekt typu Promise nahlížet jako na jakousi černou krabíčku, která je prázdná, ale jednoho dne se v ní objeví nějaká hodnota (opravdový výsledek té asynchronní funkce, která Promise vrátila). Nevíme, kdy to bude, ale můžeme k této krabíčce přidat callback a ten bude vykonán, jakmile se hodnota objeví. Tím se pomalu vracíme zpět k funkci **fetch**, která Promise používá, tedy vrací. A to hned dvakrát, protože zpracování odpovědi od serveru je rozděleno na dva kroky: když po síti dorazí hlavičky HTTP odpovědi a když následně dorazí i celé tělo:

```
function onError(e) {  
  console.error("Chyba při získávání dat", e);  
}  
  
function onBody(data) {  
  console.log(data);  
}
```

```
function onResponse(response) {
  response.json().then(onBody, onError);
}

fetch(url).then(onResponse, onError);
```

Funkce **onResponse** je vykonána, jakmile dorazí hlavičky odpovědi. Jejím parametrem je objekt odpovědi, jehož metody dovolují přístup k získaným datům. Metodou **json()** vyžádáme tělo ve formátu JSON, ale protože veškerá data nemusela zatím dorazit, jedná se opět o asynchronní funkci. Její vrácené Promise pak pomocí **then** předáme callback **onBody**, který dostane veškerá data vrácená ze serveru.

Za povšimnutí stojí, že metodě **then** můžeme předat i druhý callback (v našem případě funkci **onError**). Ten bude vykonán, pokud asynchronní funkce nedokáže splnit svůj úkol. Platí tedy, že v případě úspěchu bude vykonán první callback a v případě neúspěchu ten druhý. Zde vidíme jasnou výhodu Promises v porovnání s předáváním jednoho callbacku přímo asynchronní funkci. U něj bychom museli případný neúspěch rozhodovat na základě parametru, se kterým bude vykonán.

V praxi se s takovým použitím Promises, resp. metody **then**, ale většinou nesetkáme. Přechod od callbacků k Promises byl dobrý evoluční krok, ale v roce 2017 se JavaScript dočkal ještě razantnějšího vylepšení práce s CPS: klíčových slov **async** a **await**. Jejich použití je úzce spjaté s Promises a dovoluje nám poskládat zdrojový kód tak, aby na první pohled působil jako synchronní. Konkrétně:

- Klíčové slovo **await** představuje alternativu k volání metody **then**. Můžeme jej napsat jako operátor před hodnotu, která je Promise. Vyjadřujeme tím, že následující řádky kódu chceme vykonat až poté, co tato Promise nabude nějaké hodnoty – stejně jako kdybychom je zabalili do malé anonymní funkce a tu předali jako callback metodě **then**.
- To ovšem znamená, že právě definovaná funkce (ta, ve které použijeme **await**) bude asynchronní; některé její řádky (ty, co následují za **await**) se vykonají až poté, co funkce skončí. Proto ji musíme označit klíčovým slovem **async** a díky tomu bude její hodnota automaticky převedena na Promise.

Nejlépe to pochopíme úpravou minulé ukázky na `async/await`:

```
async function search(query) {
  try {
    let url = `/search?query=${encodeURIComponent(query)}`;
    let response = await fetch(url);
    let songs = await response.json();
  } catch (e) {
    console.error("Chyba při získávání dat", e);
  }
}
```

Méně zkušený programátor by téměř nepoznal, že se jedná o asynchronní kód, ve kterém se objevuje hned několik proměnných typu `Promise`. Výsledek je snadno čitelný, ušetřili jsme několik anonymních funkcí a zároveň dokážeme dobře zpracovat případné chyby, neboť součástí implementace klíčového slova **`await`** je i korektní spolupráce s konstrukcí **`try-catch`**.

Na závěr podkapitoly si ještě ujasněme, jak s touto funkcí **`search`** pracovat. Před její definicí se objevuje klíčové slovo **`async`**, což čtenáři dává garanci, že funkce vrátí `Promise`. Můžeme ji tedy volat těmito způsoby:

1. Úplně běžně, pokud nás její návratová hodnota nezajímá:

```
function onSubmit(e) {
  e.preventDefault();
  let query = form.querySelector("[name=query]").value;
  search(query);
}
```

2. Pomocí metody **`then`**, pokud chceme vykonávat nějakou aktivitu, až když bude hledání hotovo:

```
function onDone() {
  console.log("Hledání hotovo");
}

function onSubmit(e) {
  e.preventDefault();
```

```

    let query = form.querySelector("[name=query]").value;
    search(query).then(onDone);
  }

```

3. Ekvivalentně pomocí **await**, což s sebou ovšem nese povinnost označení našeho posluchače jako **async**:

```

async function onSubmit(e) {
  e.preventDefault();
  let query = form.querySelector("[name=query]").value;
  await search(query);
  console.log("Hledání hotovo");
}

```

Tato poslední varianta si zaslouží speciální pozornost. Není na ní nic špatného, ale používáním asynchronních posluchačů se vystavujeme riziku nepozornosti, která může vyústit v zákeřnou chybu. Vzpomeňme na úlohu z předchozí kapitoly, ve které jsme ověřovali dostupnost uživatelského jména. Čtenář by po pročetí části o **async/await** mohl chtít zmíněnou kontrolu implementovat v rámci události **submit** takto:

```

async function checkUsername() {
  // ...
}

async function onSubmit(e) {
  let available = await checkUsername();
  if (!available) { e.preventDefault(); }
}

form.addEventListener("submit", onSubmit);

```

Umíme vysvětlit, proč tento nevině vypadající kód nemůže fungovat? Pro snazší pochopení funkci **onSubmit** ekvivalentně přepíšeme na kód bez klíčových slov **async** a **await**, tj. pomocí **then**:

```

function onSubmit(e) {
  checkUsername().then(available => {
    if (!available) { e.preventDefault(); }
  });
}

```

```
});  
}
```

Kdy dojde k vykonání vnitřní anonymní arrow funkce? Příliš pozdě; až dávno po skončení posluchače **onSubmit**. Tou dobou už je ovšem příliš pozdě volat metodu **e.preventDefault()**, tuto možnost jsme měli jen během vykonávání posluchače. Plyne z toho poučení: *Zabránit výchozímu zpracování události (např. odeslání formuláře) můžeme jen v rámci synchronního kódu posluchače. Jakmile je naše kontrola asynchronní, nelze pomocí ní odeslání podmíněně ovlivňovat.*

SPA administrační systém

Úloha

V rámci webových stránek nabízíme možnost přidávání komentářů. Komentáře jsou zobrazovány až po jejich schválení správcem webu. Proto potřebujeme administrační systém, který vypíše komentáře a k nim jednotlivé možnosti:

- nové komentáře lze *schválit* nebo *smazat*,
- již schválené komentáře lze *smazat*.

Serverová strana je již připravena. Poskytuje následující API:

- výpis komentářů: **GET** `/comments`, každý komentář je objekt s vlastnostmi **id**, **author**, **text** a **approved** (typu **boolean**)
- schválení komentáře: **POST** `/comments/id-komentare/approve`
- smazání komentáře: **DELETE** `/comments/id-komentare`

Řešení

```
<!-- kapitola-7.html -->
<h2>Nové komentáře</h2>
<ul id="new"></ul>

<h2>Schválené komentáře</h2>
<ul id="approved"></ul>

<script src="kapitola-7.js"></script>
```

```
// kapitola-7.js
function deleteComment(id) {
  return fetch(`/comments/${id}`, {method: "DELETE"});
}
```

```
function approveComment(id) {
  return fetch(`/comments/${id}/approve`, {method: "POST"});
}

function buildButton(label) {
  let button = document.createElement("button");
  button.textContent = label;
  return button;
}

function buildComment(comment) {
  let node = document.createElement("li");
  node.textContent = `${comment.author}: ${comment.text}`;

  if (!comment.approved) {
    let approveButton = buildButton("Schválit");
    node.append(approveButton);
    approveButton.addEventListener("click", async e => {
      await approveComment(comment.id);
      loadComments();
    });
  }

  let deleteButton = buildButton("Smazat");
  node.append(deleteButton);
  deleteButton.addEventListener("click", async e => {
    await deleteComment(comment.id);
    loadComments();
  });

  return node;
}

async function loadComments() {
  let response = await fetch("/comments");
  let comments = await response.json();
}
```

```

let newNodes = comments.filter(c => !c.approved).map(buildComment);
document.querySelector("#new").replaceChildren(...newNodes);

let approvedNodes = comments.filter(c => c.approved)
  .map(buildComment);
document.querySelector("#approved")
  .replaceChildren(...approvedNodes);
}

loadComments();

```

Vzorové řešení vzniklo kombinací technik z minulých kapitol. Se serverem komunikujeme pomocí funkce **fetch** a používáme úsporný zápis *async/await* (minulá podkapitola pro profíky). Jednotlivé komentáře do stránky vypisujeme pomocí funkcionální iterace (minulá podkapitola pro koumáky), kdy metodou **filter** získáme správné komentáře, metodou **map** z nich vytvoříme HTML prvky a nakonec je vložíme do stránky hromadně díky DOM-metodě **replaceChildren**.

Poznamenejme, že funkce **fetch** dovoluje vytvářet libovolné druhy HTTP požadků. Pokud se jedná o metodu odlišnou od výchozí **GET**, uvedeme ji v druhém parametru **fetch**, který slouží jako konfigurační objekt.

Nejsložitější je funkce **buildComment**, která je zodpovědná jak za výpis jednoho komentáře, tak za definici administrační funkcionality. Vytváří jedno až dvě tlačítka a přidává jim požadovanou funkcionality. Používá k tomu malé anonymní asynchronní arrow funkce – běžný jev v moderním JavaScriptu.

Řešení je sice krátké, ale není příliš propracované. Namísto tradičního souhrnu *co jsme se naučili* může zvědavý čtenář promyslet tři místa, ve kterých by se dalo vylepšit:

1. Po provedení nějaké operace s komentářem voláme funkci **loadComments**, abychom načetli a zobrazili nový stav. To je bezpečný přístup (zobrazujeme vždy data v tom stavu, v jakém jsou na serveru), ale představuje úplně zbytečný přenos většiny dat ze serveru na klienta. Operace s jedním komentářem nemá na ty ostatní vliv, a přesto je pokaždé načítáme a vyrábí-

me znova. Lepší by bylo, kdyby smazání komentáře jen odstranilo jeho HTML reprezentaci; schválení by jej pak mohlo přesunout z jednoho výčtu do toho druhého. Těch ostatních se aktivita vůbec nemusí dotýkat.

2. Mazání je nevratná operace. Po kliku na tlačítko by bylo dobré nechat správce potvrdit, že si smazání opravdu přeje. Toho můžeme docílit snadno např. vestavěnou funkcí **confirm**, která zobrazí potvrzovací dialog.
3. Kód neuvažuje speciální případy, které by bylo záhodno ošetřit:
 - když selže volání serveru (všechny tři metody)
 - když v rámci **loadComments** nedostaneme žádné neschválené komentáře
 - když v rámci **loadComments** nedostaneme žádné schválené komentáře

Implementace výše uvedených návrhů nevyžaduje žádné nové znalosti, takže si je čtenář může zkusit za domácí úkol. My se zatím půjdeme podívat na něco nového.

Zelenáči: event delegation

Při procházení vzorového řešení nás může napadnout, že při větším počtu diskuzních příspěvků vytváříme značné množství JavaScriptových funkcí. Jsou to právě ty zmiňované malé arrow funkce, které fungují jako posluchače událostí na administračních tlačítkách. Jejich jediný účel je držet v uzávěře ID komentáře, se kterým se má po kliknutí pracovat. Pojdme se podívat na oblíbenou techniku, která nám dovoluje použít jen jeden posluchač (nezávisle na počtu komentářů či tlačítek).

Klíčové je v tuto chvíli pozorování, že HTML prvky vytváří stromovou strukturu a že uživatelská interakce (tj. zdroj vzniku události) se proto může týkat více uzlů naráz. Konkrétněji, když máme například odstavec a v něm odkaz, na který klikneme, *kliknuli jsme na odstavec*?

```
<p>
  Toto je odstavec. V něm je
  <a> odkaz, </a>
  na který klikneme. Bylo kliknuto (také) na odstavec?
</p>
```

Na tuto otázku je odpověď jednoznačná: **ano**, kliknutí na libovolný HTML prvek ve stránce zároveň znamená, že bylo kliknuto i na jeho rodiče (a všechny jeho předky). Pokud bychom tedy v našem administračním systému přidali posluchač události **click** kupříkladu na prvek **document.body** (odpovídá HTML značce **<body>**), bude vykonán při kliknutí na kteroukoliv část dokumentu, tedy i tlačítka.

Této technice se říká **event delegation** (český překlad *delegování událostí* se příliš neujal). Použijeme při ní jediný posluchač na společném rodiči všech prvků, které nás zajímají. Když pak nastane událost, musíme rozpoznat, co to pro nás znamená. Zpravidla prozkoumáme objekt události (parametr posluchače), z něj zjistíme, kde ve stránce událost nastala, a podle toho se zařídíme. Je to ideální místo pro použití vlastnosti **target**, o které jsme se dozvěděli ve čtvrté kapitole. Začneme tedy s úpravou vzorového řešení:

```
function onClick(e) {
  console.log("Kliknuto na", e.target);
}

function buildComment(comment) {
  let node = document.createElement("li");
  node.textContent = `${comment.author}: ${comment.text}`;

  if (!comment.approved) {
    let approveButton = buildButton("Schválit");
    node.append(approveButton);
  }

  let deleteButton = buildButton("Smazat");
  node.append(deleteButton);

  return node;
}

document.body.addEventListener("click", onClick);
```


Funkce **buildComment** se příjemně zjednodušila, protože už neobsahuje práci s událostmi. Tu bychom rádi provedli uvnitř **onClick**, ale zatím nevíme jak. Sice poznáme, na které tlačítko bylo kliknuto, ale:

- nevíme, o který komentář jde;
- nevíme, jestli bylo kliknuto na schválení nebo na smazání.

K dispozici ovšem máme HTML prvek, na kterém událost nastala. Kdybychom mu potřebné informace předali jako vlastnosti či atributy, mohli bychom je z něj získat v posluchači **onClick**. Při návrhu takového řešení budeme následovat dvě doporučení:

1. Přidávání nových vlastností HTML prvkům není dobrý nápad, protože se vystavujeme riziku, že námi zvolený název bude kolidovat s nějakou existující vlastností (co hůře: taková zatím nemusí existovat, ale časem vznikne, a tím se náš kód zničeho nic rozbije!). Místo toho budeme využívat atribut **data-***, který je určený právě pro uživatelská data. Pracovat s ním můžeme pohodlně pomocí DOM vlastnosti **dataset**. Například JavaScriptový přístup k vlastnosti **document.body.dataset.a** odpovídá čtení či zápisu uživatelského HTML atributu **<body data-a="...">**.
2. Víme, že pro provedení aktivity po kliknutí musíme získat dva údaje: ID komentáře a druh akce. Mohli bychom si oba dva uložit ke každému tlačítku, ale tím bude ID komentáře u tlačítek uloženo nadbytečně. S ohledem na stromovou strukturu lze ID komentáře nastavit například jejich společnému rodiči, tedy značce ****.

Funkce **buildButton** a **buildComment** proto upravíme takto:

```
function buildButton(label, action) {
  let button = document.createElement("button");
  button.textContent = label;
  button.dataset.action = action;
  return button;
}

function buildComment(comment) {
  let node = document.createElement("li");
  node.dataset.id = comment.id;
```

```

node.textContent = `${comment.author}: ${comment.text}`;

if (!comment.approved) {
  let approveButton = buildButton("Schválit", "approve");
  node.append(approveButton);
}

let deleteButton = buildButton("Smazat", "delete");
node.append(deleteButton);

return node;
}

```

Vzniklé HTML je dostatečně *označené*, abychom s ním mohli dokončit delegování událostí. Ve funkci **onClick** zjistíme, zdali došlo ke kliknutí v prvku s atributem **data-action** a zároveň v prvku s atributem **data-id**:

```

async function onClick(e) {
  let idNode = e.target.closest("[data-id]");
  let actionNode = e.target.closest("[data-action]");
  if (!idNode || !actionNode) { return; }

  let id = idNode.dataset.id;
  switch (actionNode.dataset.action) {
    case "approve":
      await approveComment(id);
      loadComments();
      break;
    case "delete":
      await deleteComment(id);
      loadComments();
      break;
  }
}

```

V takto upravené funkci je několik novinek. První je použití nové DOM metody **closest**, která vrací prvního předka daného prvku vyhovujícího zadanému selektoru. Příjemnou vlastností této metody je skutečnost, že hledání začíná na

prvku samém (tj. na **e.target**), což pokryje případy kliknutí na tlačítko. V tomto smyslu bychom mohli rovnou psát **e.target.dataset.action**, ale tím by náš kód přestal fungovat, jakmile bychom do tlačítka vložili další HTML obsah – například obrázek. Pak by při kliknutí na obrázek **e.target** odpovídal HTML značce `` a na ní bychom atribut **action** hledali marně.

Pokud nenalezneme **idNode** nebo **actionNode**, funkci rovnou opustíme. Je to opět příkladem přístupu *return early* a pokrýváme tak scénáře klikání kamkoliv mimo akční tlačítka. Nově se dále setkáváme s konstrukcí **switch**, která kód větví do několika bloků podle hodnoty testovaného výrazu.

Na závěr podkapitoly ještě zmiňme, že hodnoty užívané v rozhraní **dataset** jsou vždy automaticky převáděny na řetězce. V právě řešené úloze to nevadí (hodnoty ID na server posíláme jako součásti URL, tj. taktéž jako řetězce), ale pokud bychom serveru posílali data v nějakém typovaném formátu, možná by bylo nutné přetypování. Otázce převodu řetězce na číslo se proto věnuje nadcházející podkapitola pro profíky.

Koumáci: bublání a zachytávání

V minulé podkapitole jsme se přesvědčili o tom, že událost *click* je možné zachytit i na libovolném rodiči toho HTML prvku, na kterém bylo kliknuto. U jiných druhů událostí bohužel situace takto snadná být nemusí. V minulosti jsme se například věnovali události **submit**, kterou uživatel vyvolal interakcí s tlačítkem. A taková událost nastává na HTML prvku `<form>` – nikoliv na tlačítku a nikoliv na rodičích formuláře. Znamená to, že události dělíme do dvou hlavních kategorií: ty, které krom původního prvku nastávají i na všech rodičích (říkáme o nich, že *bublají*), a ty odehrávající se jen na jednom HTML prvku (říkáme, že *nebublají*). Toto pozoruhodné názvosloví snad vyjasníme za malou chvíli, ale nejprve se podívejme na jeden další scénář, kdy se nám bublající událost může hodit.

Představme si, že do stránky umístíme kontaktní formulář, který je ovšem na začátku skrytý a uživatel jej může zobrazit kliknutím na tlačítko. Zatím je to hračka:

```
<form hidden> ... </form>
<button> Kontaktovat </button>
```

```
let form = document.querySelector("form");
let button = document.querySelector("button");
button.addEventListener("click", e => form.hidden = false);
```

Situace se může zkomplikovat, když pak dostaneme za úkol *skrýt formulář, jakmile uživatel klikne mimo něj*. Protože událost **click** patří mezi bublající, můžeme přidat posluchač třeba na celý dokument. Tím se dozvíme o kliknutí *kdekoliv ve stránce* a formulář skryjeme.

```
function hideForm(e) {
  // opravdu vždy?
  form.hidden = true;
}
document.addEventListener("click", hideForm);
```

Toto řešení ovšem není funkční, neboť skrývá formulář při kliku kamkoliv, tedy i do formuláře. Naštěstí máme dva dobré způsoby, jak tomu zabránit:

1. Víme, že vlastnost **e.target** odpovídá tomu HTML prvku, na kterém událost nastala. Pokud dokážeme ověřit, že se tento nachází někde uvnitř formuláře (nebo je to přímo formulář), můžeme za takové situace kliknutí ignorovat. Takový test (patrně pomocí metody **closest**) by odpovídal příkladu z předchozí podkapitoly pro zelenáče.
2. Prohlížeč vykonává posluchače bublajících událostí v pevně definovaném pořadí *odspoda nahoru*, tedy od cílového prvku směrem ke kořeni stromu dokumentu. Když klikneme na HTML prvek, nejprve se vykoná posluchač přidáný přímo na něj (pokud nějaký je), pak na jeho rodiči, pak na rodiči jeho rodiče... až k poslednímu místu, tj. k celé proměnné **document**. Tento proces můžeme zastavit v rámci posluchače, který přidáme na formulář.

Pojďme si vyzkoušet druhé řešení. Stačí nám k němu znát novou metodu **stopPropagation**, která je součástí objektu události. Jejím účelem je zastavit proces bublání:

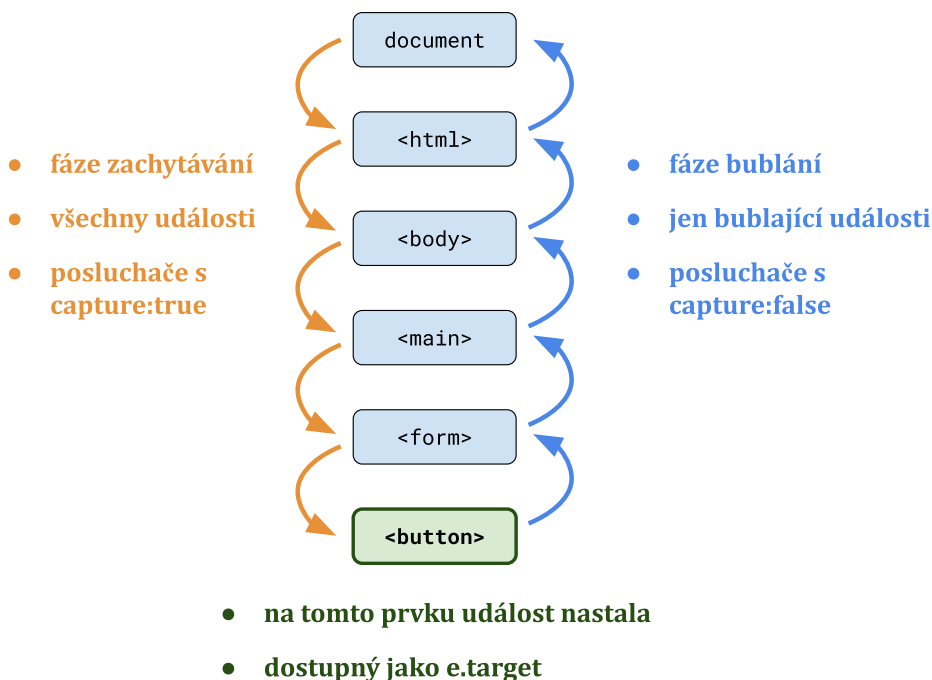
```
document.addEventListener("click", e => form.hidden = false);
form.addEventListener("click", e => e.stopPropagation());
```

Takový přístup můžeme popsat slovy *dokument se nedozví o tom, že bylo kliknuto na formulář*. Při kliku kamkoliv (do formuláře i mimo něj) se vykoná jen jeden ze dvou výše uvedených posluchačů.

Na úloze se skrýváním formuláře je vidět, že *bublání* je silná technika. Bohužel si ale vzpomínáme, že některé události nebublají. Nemusíme se však obávat, protože zpracování posluchačů pomocí bublání je ve skutečnosti jen polovina toho, jak prohlížeč na událost reaguje. Před vykonáváním našich posluchačů totiž ještě nastane okamžik, kterému se říká *zachytávání* (anglicky *capture*). Během něj prohlížeč projde stromem dokumentu *shora dolů* od kořene až k cílovému prvku (tedy v obráceném pořadí, než u bublání) a cestou vykoná ty posluchače právě zpracovávané události, u kterých jsme explicitně požádali o spuštění ve fázi zachytávání. K tomu slouží třetí, nepovinný parametr metody **addEventListener**. Pokud máme o zachytávání zájem, musíme jej nastavit buď na **true**, nebo na objekt obsahující **capture: true**:

```
// dvě shodné varianty
document.addEventListener("focus", console.log, true);
document.addEventListener("focus", console.log, {capture: true});
```

Celý proces zpracování události si můžeme snadněji představit pomocí obrázku:



Obrázek: Životní cyklus události

Pojďme si shrnout důležité kroky v životě zpracovávané události.

- Jakmile nastane DOM událost, prohlížeč vytvoří objekt události a začne vykonávat posluchače. Nejprve ve fázi zachytávání, pak ve fázi bublání.
- Posluchače ve fázi zachytávání musí být explicitně označeny a jsou vykonávány v pořadí **shora dolů**.
- Posluchače ve fázi bublání jsou vykonávány v pořadí **odspoda nahoru**. Pokud událost neublá, vykoná se jen posluchač na cílovém HTML prvku a žádný další.
- Procesy zachytávání i následného bublání lze v kterémkoliv posluchači zastavit voláním `stopPropagation()`.

U většiny JavaScriptových úloh si vystačíme s bubláním. Zachytávání se nám může hodit jen ve dvou situacích:

1. Když potřebujeme prohodit pořadí dvou posluchačů stejné události na různých HTML prvcích.

2. Když se potřebujeme na rodičovském prvku dozvědět o nebublající události jeho potomka.

Profíci: o řetězcích a číslech

Řetězce i čísla patří mezi základní datové typy, se kterými v JavaScriptu pracujeme. V páté kapitole jsme se ujistili v tom, že nejlepší je, když s nimi pracujeme odděleně: řetězce porovnáváme s řetězci a čísla s čísly. Čas od času se ale dostaneme do situace, kdy jednu konkrétní hodnotu potřebujeme reprezentovat jednou jako číslo a podruhé jako řetězec. Může to být příklad rozhraní **dataset** z podkapitoly pro zelenáče, ale třeba i nastavování hodnot HTML prvku `<input>` pomocí vlastnosti **value**. V těchto situacích dochází k implicitnímu (automatickému) převodu hodnoty na řetězec.

Takový proces nevyžaduje detailnějšího prozkoumávání, protože zpravidla funguje tak, jak očekáváme. Pokud bychom potřebovali číslo na řetězec převádět ručně, máme k tomu několik nástrojů:

- univerzální funkci **String**, která převede libovolný datový typ na řetězec;
- metodu **toString**, kterou mají mimo jiné všechna čísla a u které můžeme parametrem specifikovat, jakou číselnou soustavu použít (při neuvedení se použije desítková);
- specializované metody **toFixed** a **toPrecision**, které dovolují různé formy zápisu desetinných čísel.

Opačný převod z řetězce na číslo je trochu složitější. Souvisí to mimo jiné s tím, že ne všechny řetězce lze na číslo převést, neboť mohou obsahovat znaky, které se v reprezentaci čísel nepoužívají. K tomuto účelu existují tři globální funkce: **Number**, **parseInt** a **parseFloat**. Podíváme se na jejich vlastnosti a rozdíly.

Především nás zajímá situace, kdy převod není možné provést, neboť řetězec obsahuje neplatný znak. Tehdy bude výsledkem převodu (u všech tří funkcí) speciální hodnota **NaN** (z anglického *not a number*). Pokud nemáme jistotu, že řetězec obsahuje platné číslo, měli bychom výsledek převodu zkontrolovat:

```
let id = Number(input.value);
```

```
if (Number.isNaN(id)) {
  console.error("Hodnota není číslo");
}
```

Specializovaná testovací funkce **isNaN** je zde nutná, neboť prosté porovnání s **NaN** by selhalo. Je to způsobeno tím, že exotická hodnota **NaN** se z definice ničemu nerovná, ani sama sobě. V ukázce jsme použili převodní funkci **Number**, která je ve většině případů správnou volbou. Jejím charakteristickým rysem je, že pokud narazí na neplatný znak, okamžitě vrátí **NaN**, i kdyby mu předcházely platné číslice.

Nikoliv tak zbývající dvě funkce, **parseInt** a **parseFloat**. Ty prochází řetězec od začátku, znak po znaku, a pokud narazí na neplatný znak, pokusí se vytvořit číslo z toho, co doposud zpracovaly. Jejich názvy dávají tušit, že návratovou hodnotou je celé, respektive desetinné číslo. To je trochu zavádějící, neboť v JavaScriptu je jen jeden číselný datový typ (formálně se nazývá *IEEE 754 double-precision floating-point format*). Pokud tedy v řetězci zapíšeme celé číslo, výsledky se budou rovnat. Rozdíl nastane až v okamžiku, kdy řetězec obsahuje desetinnou tečku. Funkce **parseInt** se na ní zastaví a vrátí to celé číslo, které do té doby zpracovala. Jinými slovy, ořízne z desetinného čísla jeho desetinnou část. Nejlépe si hlavní rysy těchto konverzních funkcí ukážeme na příkladech:

```
Number("42");           // 42
Number("42b");           // NaN

parseInt("42b");          // 42
parseInt("b42");          // NaN
parseInt("42") === parseFloat("42"); // true

parseInt("42.9");         // 42
parseFloat("42.9");       // 42.9
```

Pro většinu úloh si vystačíme s funkcí **Number**. Ve speciálních případech se ještě může hodit druhý parametr funkce **parseInt**, kterým říkáme, v jaké číselné soustavě je převáděný řetězec zapsán. Můžeme tak získat hodnoty zapsané například v šestnáctkové, nebo až šestatřicítkové soustavě:

```
parseInt("CAFE", 16);    // 51966
parseInt("AHOJ", 36);    // 489475
```


Tato kvalita funkce `parseInt` je ovšem zároveň i jejím prokletím, na které může neopatrný programátor doplatit například ve funkcionální iteraci. Představme si, že máme pole celých čísel zapsaných jako řetězce a chceme je převést na čísla. V tomto pomyslném poměřování máme dva závodníky. Dovedete odhadnout, jak výsledek dopadne?

```
let numbers = ["10", "10", "10"];  
  
numbers.map(Number);  
numbers.map(parseInt);
```

Aplikace funkce `Number` vrátí pole tří správných desítek. Výsledkem druhého volání je ovšem značně nezvyklé pole `[10, NaN, 2]` – a méně zkušený programátor si bude dlouho lámat hlavu nad tím, proč se správně podařilo zpracovat jen první hodnotu.

Odpověď můžeme nalézt v minulé kapitole, respektive její podkapitole pro koumáky. Připomeňme, že v rámci funkcionální iterace `map` dostává iterační funkce (zde `parseInt`) celkem tři parametry: iterovaný prvek, jeho pořadí (index) a celé pole. Nastanou tedy tato tři volání:

```
parseInt("10", 0, ["10", "10", "10"]); // 10  
parseInt("10", 1, ["10", "10", "10"]); // NaN  
parseInt("10", 2, ["10", "10", "10"]); // 2
```

Třetí parametr je funkcí `parseInt` ignorován, druhý specifikuje číselnou soustavu. Nula zde funguje jako *nezadáno*, použije se tedy výchozí desítková soustava. Na druhém řádku vyžadujeme použití *jedničkové soustavy*, což je sice teoreticky možné, ale v praxi zakázané – výsledek je `NaN`. A třetí řádek představuje převod čísla ve dvojkové soustavě, kde zápis `"10"` odpovídá desítkovému číslu 2.

Třídy a objekty

Úloha

Administrační systém z minulé kapitoly se rozrostl a stávající implementace nedovoluje snadné rozšiřování. Rozdělte kód do menších, samostatně funkčních celků. Použijte techniky objektově orientovaného programování.

Řešení

```
<!-- kapitola-8.html -->
<h2>Nové komentáře</h2>
<ul id="new"></ul>

<h2>Schválené komentáře</h2>
<ul id="approved"></ul>

<script type="module" src="kapitola-8.js"></script>
```

```
// kapitola-8.js
import Comment from "./comment.js";

export async function loadComments() {
  let response = await fetch("/comments");
  let comments = await response.json();

  let newList = document.querySelector("#new");
  let approvedList = document.querySelector("#approved");
  newList.replaceChildren();
  approvedList.replaceChildren();

  comments.forEach(c => {
    let comment = new Comment(c);
```

```

    (c.approved ? approvedList : newList).append(comment.node);
  });
}

```

```
loadComments();
```

```

// comment.js
import { loadComments } from "../kapitola-8.js";

export default class Comment {
  constructor(data) {
    let node = document.createElement("li");
    node.textContent = `${data.author}: ${data.text}`;

    if (!data.approved) { node.append(this.buildApproveButton()); }
    node.append(this.buildDeleteButton());

    this.id = data.id;
    this.node = node;
  }

  buildApproveButton() {
    let button = buildButton("Schválit");
    button.addEventListener("click", async e => {
      await this.approve();
      loadComments();
    });
    return button;
  }

  buildDeleteButton() {
    let button = buildButton("Smazat");
    button.addEventListener("click", async e => {
      await this.delete();
      loadComments();
    });
    return button;
  }
}

```

```

    }

    delete() {
        return fetch(`/comments/${this.id}`, {method: "DELETE"});
    }

    approve(id) {
        return fetch(`/comments/${this.id}/approve`, {method: "POST"});
    }
}

function buildButton(label) {
    let button = document.createElement("button");
    button.textContent = label;
    return button;
}

```

Kód z minulé kapitoly doznal značných změn, přesto je jeho funkcionalita stejná. Provedli jsme **refactoring**: přepracování kódu za účelem přípravy na další rozšiřování.

Třídy

O vykreslení komentáře a související interaktivitu se nově stará třída **Comment**. JavaScriptové třídy jsou klasickým mechanismem pro objektově orientované programování. Jde o koncept entit, které spolu kombinují správu dat a logiku, jež s těmito daty pracuje. Třída je pak jakási šablona, která popisuje, jak budou z ní odvozené objekty (těm se říká *instance*) fungovat. Definice a chování tříd v JavaScriptu se blíží jiným tradičním objektově orientovaným jazykům, jako je Java nebo C++.

Syntakticky je definice třídy zabalená do složených závorek a jedná se vlastně jen o výčet jednotlivých metod, kterými budou instance této třídy disponovat. Speciálně pojmenovaná metoda **constructor** bude vykonána vždy při vzniku nové instance, tj. při použití zápisu **new Comment**. Zájemcům o detailnější pochopení toho, *co to vlastně třídy jsou*, je pak určena podkapitola pro profíky.

V metodách tříd se často objevuje důležité klíčové slovo **this**. Pomocí něj můžeme odkazovat na tu instanci třídy, jejíž metoda je právě vykonávána. Snadno tak pro konkrétní komentář zavoláme nějakou jeho metodu (např. **this.approve()**, **this.delete()**), nebo přistoupíme k jeho vlastnostem (**this.id**). Klíčové slovo **this** je ve skutečnosti výrazně komplikovanější, než se při pohledu na vzorové řešení zdá, a je mu proto věnována podkapitola pro koutačky.

JS moduly

Druhá novinka je rozdělení JavaScriptu do dvou souborů. To je logický krok ve chvíli, kdy objem kódu přesáhne jistou subjektivní mez. V našem případě se jedná o hlavní soubor **kapitola-8.js** (obsahuje logiku načítání) a dále **comment.js** (obsahuje management komentářů). Aby bylo možné realizovat spolupráci mezi těmito soubory, použijeme koncept JS modulů (poprvé jsme o něm slyšeli ve druhé kapitole, v podkapitole pro profíky). To znamená tyto kroky:

1. V HTML prvku **<script>** přidáme atribut **type="module"**. Tím je soubor **kapitola-8.js** považován za JS modul a jím importované soubory taktéž.
2. Pokud chceme nějakou funkcionalitu v JS modulu nabídnout k použití, přidáme před ni klíčové slovo **export**.
3. Pokud chceme v jednom souboru přistoupit k funkcionalitě z jiného, musíme ji nejprve importovat klíčovým slovem **import**.

Pozor! Při používání JS modulů přistupuje prohlížeč o něco striktněji k atributu **src** u HTML prvku **<script>**. Toto URL nově podstupuje kontrolu *originu* (viz pátou kapitolu) a musí používat protokol HTTP, tj. zejména není možné jej načítat pseudo-protokolem **file://**. To je pro rychlý lokální vývoj komplikace, neboť se u JS modulů neobejdeme bez opravdového HTTP serveru, který bude soubory (HTML, JS a další) vydávat. Jakmile tedy začneme JS moduly používat, budeme se muset seznámit s libovolným webovým serverem. Naštěstí je takových velké množství: může to být dedikovaná aplikace (Apache, Nginx), vývojový server vestavěný do různých jazyků (PHP, Python Flask, Node.js) nebo třeba rozšíření do oblíbeného IDE.

JS moduly mohou exportovat libovolné množství *pojmenovaných věcí* (proměnných, funkcí, tříd, ...) a také jeden tzv. *výchozí (default) export*. Rozdíl mezi pojmenovanou a nepojmenovanou věcí je jen v tom, jak se k nim následně přistupuje z jiného souboru. Třída **Comment** je výchozí export (při importu pak uvádíme jen

jméno, pod kterým má být výchozí export dostupný v importujícím souboru), funkce `loadComments` je pojmenovaný export (při importu musíme uvést její název ve složených závorkách).

Všimněme si také, že naše dva soubory na sobě navzájem závisí. Soubor `kapitola-8.js` importuje třídu `Comment` z `comment.js` a naopak třída `Comment` volá funkci `loadComments` z `kapitola-8.js`. Takové situaci se říká *kruhová závislost* a může to být známkou nevhodně navržené aplikace. V našem případě to ničemu nevadí, ale i tak pojďme zvážit, jak bychom se mohli kruhové závislosti vyhnout.

Funkce `loadComments` potřebuje znát třídu `Comment`, protože sama o sobě neumí získaná data zpracovat (to je hlavním cílem refactoringu v této kapitole). Taková závislost je tedy v pořádku. Abychom kruhovou závislost rozbili, mohli bychom třídu `Comment` přepracovat tak, aby po změně komentáře nevolala `loadComments` (pak by nemusela importovat nic ze souboru `kapitola-8.js`). Jinými slovy, aby se v rámci posluchačů klikání na tlačítka jen *dalo vědět*, že došlo ke změně komentáře – a nějaká jiná komponenta by pak rozhodla, že je nutné data znovu načíst.

Jedním možným přístupem by bylo vytvoření *vlastní události*, kterou by třída `Comment` vyvolala po změně komentáře. Tuto událost může zachytit kód v hlavním modulu a zavolat `loadComments`. Takové řešení dobře zapadá do ekosystému DOM událostí, ale pro naše potřeby je možná zbytečně složitá. Třída `Comment` může o změně komentáře dát vědět i jednodušším způsobem:

```
class Comment {
  commentChanged() {}

  buildApproveButton() {
    let button = buildButton("Schválit");
    button.addEventListener("click", async e => {
      await this.approve();
      this.commentChanged();
    });
    return button;
  }
}
```

```
let comment = new Comment(data);
comment.commentChanged = loadComments;
```

Proč definice třídy **Comment** vůbec obsahuje prázdnou metodu **commentChanged**? Ze dvou důvodů. Za prvé tím čtenáři našeho kódu naznačujeme existenci této metody, dále pak prázdnou (ale existující!) metodou vyřešíme situaci, kdy by došlo ke kliknutí na tlačítko, aniž by v instanci někdo vlastnost **commentChange** přepsal na svou vlastní funkci.

Co jsme se naučili

Po vyřešení sedmé úlohy by měl čtenář chápat a ovládat:

- definici a použití JS tříd
- dělení kódu na JS moduly
- použití klíčových slov **import** a **export**

Zelenáči: odebrání posluchačů událostí

S událostmi se setkáváme téměř v každé kapitole. Ve světě objektově orientovaného programování představují jistou výzvu zejména v tom okamžiku, kdy nás *existující posluchač přestal zajímat*, tj. kdy o něj již nestojíme. Ve vzorovém řešení se objevuje třída **Comment**, která poslouchá událost kliknutí na jednotlivých tlačítkách. Během interakce s naším administračním systémem ovšem instance třídy **Comment** vznikají (zápisem **new Comment**) a následně zase zanikají (když v důsledku **loadComments** vymažeme staré položky, o paměť se časem postará garbage collector). Co se děje se všemi těmi posluchači událostí, které již nemohou nastat?

Především je nutné poznat a odlišit situace, kdy posluchač zmizí sám od sebe (prohlížeč jej odebere) a kdy musíme naopak sami prostřednictvím metody **removeEventListener** poslouchání ukončit. Vzorová třída **Comment** představuje vzájemnou provázanost HTML prvku a instance JavaScriptové třídy:

- Dokud existuje HTML prvek **<button>** s posluchačem události, musí existovat i instance **Comment**.

- Dokud existuje JS instance **Comment**, musí existovat i jí odpovídající HTML dostupné prostřednictvím vlastnosti **node**.

To znamená, že garbage collector začne pracovat teprve poté, co přestaneme používat instance třídy a zároveň z dokumentu odstraníme relevantní HTML prvky. Instance si nikam neukládáme a při nejbližším zavolání **loadComments** zaniknou i jim odpovídající HTML uzly. Obě provázané entity zmizí včetně posluchačů událostí. Nemusíme tedy dělat nic.

Komplikovanější situace by nastala, kdyby třída **Comment** použila posluchač události mimo své vlastní HTML. Mohl by to být posluchač na objektu **window** (třeba událost **keydown** odpovídající stisku klávesy nebo událost **popstate** z šesté kapitoly) či na objektu **document** (třeba událost **paste** při vložení ze schránky). Pak by garbage collector nemohl nikdy uvolnit paměť zabranou instancí **Comment**, přestože by její vlastní HTML ze stránky již dávno zmizelo. Mohla by totiž nastat zmiňovaná událost a v jejím důsledku by se vykonal posluchač instanci náležící. Čím víc komentářů bychom zobrazili, tím víc posluchačů by zůstávalo přidáných.

To je samozřejmě nešikovné, hned ze dvou důvodů. Jednak proto, že při vzniku události dojde k vykonání dávno neužitečných posluchačů, a poté proto, že tím dochází k nárůstu zabrané paměti, kterou bychom rádi uvolnili pro užitečnější účely. Tomuto jevu se říká **memory leak** a chceme se mu rozhodně vyhnout. Budeme proto muset naše instance ve vhodnou chvíli upozornit, aby své posluchače odebraly.

Pokud je posluchač pojmenovaná funkce, můžeme jej odebrat DOM-metodou **removeEventListener**:

```
function onClick(e) { ... }

document.addEventListener("click", onClick);
// a později:
document.removeEventListener("click", onClick);
```

V případě objektů ale posluchače většinou vypadají jinak – jako malé anonymní arrow funkce, které následně volají další metody objektu. Třeba takto:


```
class Comment {
  constructor() {
    document.addEventListener("click", e => this.remove());
    // tak to fungovat nebude:
    document.removeEventListener("click", e => this.remove());
  }
}
```

Proč výše uvedený kód nebude fungovat? Neboť pro úspěšné odebrání posluchače musíme metodě **removeEventListener** předat tu samou funkci, kterou jsme dříve použili pro **addEventListener**. Naše dvě malé arrow funkce sice *dělají to samé*, ale jsou to dvě různé funkce (nerovnájí se). Pojdme se proto podívat na dvě další možnosti, jak posluchače událostí odebrat.

Metoda `handleEvent`

Navykli jsme si, že druhý parametr pro **addEventListener** je funkce. To ovšem není jediná možnost: může to být také objekt, který má vlastnost **handleEvent**. Tato vlastnost je, pakliže se jedná o funkci, volána při vzniku události. Pokud bychom tedy třídě **Comment** takovou vlastnost přidali, stala by se třída jako taková posluchačem. Kód by mohl vypadat takto:

```
class Comment {
  buildApproveButton() {
    let button = buildButton("Schválit");
    button.dataset.action = "approve";
    button.addEventListener("click", this);
    return button;
  }

  buildDeleteButton() {
    let button = buildButton("Smazat");
    button.dataset.action = "delete";
    button.addEventListener("click", this);
    return button;
  }

  async handleEvent(e) {
```

```

    let actionNode = e.target.closest("[data-action]");
    switch (actionNode.dataset.action) {
      case "delete":
        await this.delete();
        loadComments();
        break;

      case "approve":
        await this.approve();
        loadComments();
        break;
    }
  }
}

```

Hodí se nám zde technika delegování událostí, protože metoda **handleEvent** se nyní věnuje všem (oběma) interakcím, které mohou nastat. Potřebujeme v ní proto odlišit, na co bylo kliknuto, tj. co se má v rámci události vykonat.

V tuto chvíli je pak snadné posluchače odebrat. Druhý parametr známe (je to instance **Comment**, tj. **this**) a potřebujeme jen znát tlačítka, na které jsme posluchač přidali. Mohli bychom tedy třídě **Comment** přidat *ukončovací* metodu, která posluchače odebere:

```

class Comment {
  destroy() {
    const s = "button[data-action]";
    let buttons = Array.from(this.node.querySelectorAll(s));
    buttons.forEach(b => b.removeEventListener("click", this));
  }
}

```

Zvědavý čtenář jistě chápe, že tuto metodu je nutné někdy zavolat. Za domácí úkol si proto může zkusit upravit funkci **loadComments** tak, aby si vytvořené instance pamatovala a při dalším volání ty předchozí nejprve *zničila* voláním metody **destroy()**.

Objekt AbortController

Velmi odlišný přístup k odebrání posluchačů představuje využití relativně nového objektu **AbortController**. Můžeme si jej představit jako krabičku s velkým červeným tlačítkem: jeho stisk slouží k přerušení některých operací. Typicky jej používáme k přerušení HTTP požadavků a nebo právě posluchačů událostí.

Při práci s objektem **AbortController** pak posluchače neodebíráme metodou **removeEventListener**, ale pomyslným stiskem onoho tlačítka. Tím lze přerušit celou řadu aktivit či posluchačů – všechny, které jsme předtím k tlačítku připojili.

V praxi to můžeme zkusit třeba takto:

```
class Comment {
  constructor() {
    this.abortController = new AbortController();
  }

  buildApproveButton() {
    let signal = this.abortController.signal;

    let button = buildButton("Schválit");
    button.addEventListener("click", async e => {
      await this.approve();
      loadComments();
    }, {signal});
    return button;
  }

  destroy() {
    this.abortController.abort();
  }
}
```

Instance **abortController** je naše červené tlačítko; jeho vlastnost **signal** je pak ten neviditelný drát, kterým je spojeno s jedním či více posluchači. K tomu jsme metodě **addEventListener** museli přidat třetí parametr. Je jím konfigurační objekt (už jsme o něm slyšeli v minulé kapitole, v podkapitole o zachytávání událostí).

V ukázce výše jsme použili **šikovnou syntaktickou zkratku**, která mohla důkladné čtenáře zmást. Jedná se o zápis **{signal}**. Jde o běžnou tvorbu objektu, ve kterém definujeme klíč se stejným názvem a hodnotou, jako je uvedená proměnná. Pokud často tvoříme takové slovníky, můžeme vhodně pojmenovanými proměnnými ušetřit místo:

```
let age = 42;
let name = "Jiří";
let person = {age, name};
// stejné jako {age:age, name:name}
```

Použití **AbortController** je velmi výhodné tam, kde chceme naráz odebrat více posluchačů. Nemusíme si totiž nikde pamatovat ani předané posluchače (funkce či objekt s metodou **handleEvent**), ani HTML prvky, na které jsme je přidali (tlačítka). Stačí mít jen po ruce jedinou instanci **AbortController** a zavolat její metodu **abort()**.

Koumáci: klíčové slovo **this**

Naše setkání s klíčovým slovem **this** v rámci třídy bylo přímočaré. Prohlédněme si ovšem tento kód:

```
class Comment {
  constructor(data) {
    this.id = data.id;
    this.node = document.createElement("button");
    this.node.addEventListener("click", this.onClick);
  }

  onClick(e) {
    alert(this.id);
  }
}
```

```
}
}
```

Při kliknutí na tlačítko zjistíme, že vypisovaná hodnota **this.id** je **undefined** – a že **this** neodpovídá instanci **Comment**, ale tlačítku samotnému. Čím to?

V JavaScriptu je hodnota **this** uvnitř funkce určena nejen tím, jak funkci definujeme, ale především tím, jak ji voláme. Jedná se fakticky o další, implicitní lokální proměnnou, která je dostupná ve většině funkcí a metod. Její hodnota je ve chvíli volání určena řadou pravidel. Můžeme je rozdělit do několika kategorií:

1. Funkce volaná jako metoda objektu, tj. při volání je vlevo od jejího názvu objekt oddělený tečkou: hodnotou **this** bude objekt vlevo od tečky. To odpovídá přístupu v jiných objektově orientovaných jazycích.

2. Funkce předaná jako callback, tj. vykonaná v rámci jiné funkce. Zde je hodnota **this** v kompetenci toho, kdo callback volá. Za zmínku stojí tyto případy:

- Pokud je funkce posluchač události, lokální proměnná **this** v ní bude objekt (zpravidla HTML prvek), na který jsme posluchač přidali – ten vlevo od tečky při volání **addEventListener**. K této hodnotě se můžeme dostat také pomocí vlastnosti **currentTarget** objektu události (pozor na drobný rozdíl v porovnání s **target**, který odpovídá prvku, na kterém událost vznikla, nikoliv na kterém byla zachycena).
- Pokud je funkce použita v rámci funkcionální iterace (tedy je prvním parametrem pro **forEach**, **map**, **filter** a další), můžeme její hodnotu **this** určit jako druhý parametr iterační metody.

3. Speciální případy:

- Pokud je funkce vykonána prostřednictvím nepřímého volání metodami **call**, **apply** nebo **bind**, bude **this** rovno prvnímu parametru těchto metod. Tento mechanismus pochází z historických dob JavaScriptu a dnes se příliš nepoužívá.
- Pokud je funkce vykonána prostřednictvím klíčového slova **new**, bude **this** nově vytvářená instance. V této kapitole si klíčové slovo **new** ukážeme v rámci tříd, ale nadcházející podkapitola vysvětlí, že třídy jsou ve skutečnosti funkce.

- Arrow funkce jsou z těchto pravidel zcela vyjmuty: lokální proměnná **this** v nich neexistuje. Pokud v arrow funkci **this** použijeme, bude to **this** převzaté z okolního scope (stejně jako jakákoliv jiná proměnná dostupná prostřednictvím uzávěry).

Když funkci voláme jen zapsáním jejího názvu a kulatých závorek, nedává použití **this** v jejím těle smysl. I pro tento případ je ovšem chování **this** specifikováno: bude to buď **undefined**, nebo globální objekt (v prohlížeči **window**). Rozhoduje se o tom podle toho, zdali se nacházíme v tzv. *strikním režimu* (např. v rámci JS modulů – více o tom ve čtrnácté kapitole), či nikoliv.

V praxi je pro nás **this** užitečné zejména v první kategorii, kde s ním pracujeme intuitivně jako s instancí, k jejímž metodám a vlastnostem přistupujeme. Docela dobře si však vystačíme i s obvyčejnými objekty:

```
function logThis() {
  console.log(this);
}

let a = { logThis };
let b = { logThis };

a.logThis == b.logThis; // true
a.logThis();             // a
b.logThis();             // b
logThis();               // window / undefined
```

Objekty (slovníky) **a** i **b** mají jen jednu vlastnost ("**logThis**"), jejíž hodnotou je funkce. Jedná se o tu samou funkci, kterou oba objekty sdílí. Hodnota **this** je následně určena při volání.

Hlavní komplikace pak plynou ze situací z druhé kategorie, tj. z případů, kdy je funkce předána jako callback. Její součástí (při předání) není hodnota **this**, takže se pak můžeme dočkat nepříjemných překvapení:

```
class Comment {
  constructor(data) {
    this.id = data.id;
```

```

    let node = document.createElement("li");
    node.addEventListener("click", this.onClick);

    setTimeout(this.onTimeout, 5000);
  }

  onClick() { alert(this.id); }
  onTimeout() { alert(this.id); }
}

```

V ukázce výše nebude fungovat ani **onClick**, ani **onTimeout**. Při kliknutí bude **this** rovno HTML prvku ****, za pět vteřin bude **this == undefined**. Skutečnost, že callback není s **this** žádným způsobem svázán, můžeme pochopit například takovouto úvahou:

```

class Comment {
  constructor(data) {
    this.id = data.id;

    let f = this.onTimeout;
    f();           // this = undefined
    setTimeout(f, 5000); // this = undefined
  }

  onTimeout() { alert(this.id); }
}

```

Typickým způsobem, jak předat hodnotu **this** zároveň s callbackem, je vytvoření uzávěry. Můžeme to udělat buď ručně, nebo pomocí arrow funkcí, jejichž specifické pravidlo s **this** je pro nás užitečné:

```

class Comment {
  constructor(data) {
    this.id = id;

    // varianta 1: vlastní uzávěra
    let that = this;
    setTimeout(function() { that.onTimeout(); }, 5000);
  }
}

```

```

    // varianta 2: arrow funkce nemají vlastní `this`
    setTimeout( () => this.onTimeout(), 5000 );
  }

  onTimeout() { alert(this.id); }
}

```

Profíci: prototypová dědičnost

Při objektově orientovaném programování modelujeme řešenou úlohu pomocí *objektů* – entit, které spolu sdružují data a zároveň funkce, jež s těmito daty pracují. Na příkladu třídy **Comment** vidíme data uložená ve vlastnostech **node** a **id**, zatímco metody (tedy funkce) odpovídají například vlastnostem **approve** nebo **buildDeleteButton**. K vytvoření objektu s vlastnostmi a funkcemi ale třídy nepotřebujeme. Úplně nejjednodušeji bychom mohli objekt vytvořit třeba takto:

```

function createCommentObject(data) {
  let node = document.createElement("li");

  let obj = {
    node, // zkrácený zápis, viz podkapitulu pro zelenáče
    id: data.id,

    approve() {
      // ...
    },

    buildDeleteButton() {
      // ...
    }
  };

  return obj;
}

```


Byli bychom s takovouto realizací objektově orientovaného programování spokojeni? Máme zde proměnnou, která v sobě drží potřebná data a zároveň s nimi dokáže pracovat pomocí metod. Moc dobrý objekt to ale není. Naše hlavní výhrady jsou dvě:

1. Když budeme takových objektů vytvářet více (což chceme, neboť ze serveru získáme data s mnoha komentáři), musíme pro každý znovu definovat jeho metody. To je pracné a hlavně zbytečné, protože všechny naše komentáře mají metody identické. Proto bychom raději, kdyby objekty své metody sdílely.
2. Nemáme k dispozici žádný mechanismus *dědičnosti*, tj. způsob, jak pomocí existujícího objektu vystavit nový s přidanou funkcionalitou.

V JavaScriptu se pro řešení obou výhrad historicky používá koncept, který pochází z jazyka Self a říká se mu **prototypová dědičnost**. Jeho podstata je triviální a lze ji shrnout do krátké definice: *Nově vytvořenou proměnnou můžeme svázat s existujícím objektem. Pokud v této proměnné přistoupíme k neexistující vlastnosti, namísto hodnoty `undefined` se vrátí stejně pojmenovaná vlastnost z provázaného objektu.* Znamená to, že tento objekt představuje jakousi zálohu pro případ, že v odvozené proměnné potřebná vlastnost schází. Na jednoduchém příkladu:

```
let data1 = {
  name: "Jiří",
  age: 42
}

let data2 = Object.create(data1);
data2.name; // "Jiří"
data2.age;  // 42

data1.name = "Eva";
data2.age = 10;

data2.age;    // 10
data2.name;   // "Eva"
```

Funkce **Object.create** je zde klíčová. Pomocí ní můžeme vytvořit zmiňovanou *prototypovou vazbu*, která říká, že neexistující vlastnosti v **data2** budou dohledány v **data1**. Pro úplnost dodejme, že objekt **data1** označujeme jako *prototyp* objektu **data2**. Připomíná jeho *šablonu* či *vzor*.

V ukázce výše si povšimněme těchto dvou míst:

- Objekt **data2** sice zprvu vlastnost **age** nemá, ale jakmile mu ji nastavíme (zde na hodnotu **10**), tak při dalším použití už se použije tato.
- Objekt **data2** vznikl v době, kdy objekt **data1** obsahoval jméno **Eva**. Objekt **data1** jsme pak pozměnili a tato úprava se projevila zpětně i v objektu **data2**. Je tedy vidět, že **data2** není obyčejná kopie **data1**, ale že jsou spolu skutečně provázány.

Využijeme toto chování k vylepšení našich komentářů, zatím stále bez tříd:

```
let commentPrototype = {
  approve() {
    // ...
  }

  buildDeleteButton() {
    // ...
  }
}

function createCommentObject(data) {
  let node = document.createElement("li");
  let obj = Object.create(commentPrototype);

  obj.node = node;
  obj.id = data.id;
  return obj;
}
```

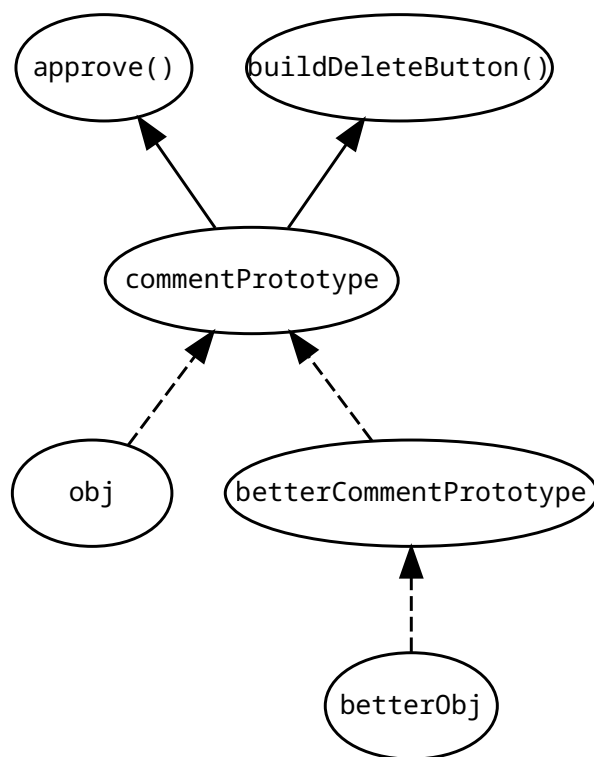
Metody objektu komentáře jsou nyní obsaženy v proměnné **commentPrototype**, a jsou tedy definovány jen jednou (a prostřednictvím prototypové dědičnosti sdíleny všemi objekty, které vrátí funkce **createCommentObject**). Zároveň jsme

otevřeli dveře ke zmíněné implementaci dědičnosti. Pokud chceme vytvořit rozšířený komentář, připravíme si nejprve jeho prototypový objekt a do něj vložíme rozšířenou funkcionalitu (zde například metoda pro kontrolu pravopisu):

```
let betterCommentPrototype = Object.create(commentPrototype);
betterCommentPrototype.checkSpelling = function() {
  // ...
}

function createBetterCommentObject(data) {
  let node = document.createElement("li");
  let betterObj = Object.create(betterCommentPrototype);

  betterObj.node = node;
  betterObj.id = data.id;
  return betterObj;
}
```



Obrázek: Dědičnost pomocí prototypové vazby (čárkovaně)

Všimněme si, že takto vylepšený komentář nabízí kontrolu pravopisu, ale zároveň i všechny metody běžného komentáře. To proto, že jeho neexistující vlastnost (třeba **approve**) je nejprve neúspěšně hledána v **betterCommentPrototype**, kde neexistuje, a proto je dále hledána v prototypu tohoto objektu (a úspěšně nalezena v **commentPrototype**). Vytvořili jsme dvě prototypové vazby za sebou, tzv. *prototype chain*.

S takovým objektově orientovaným přístupem ovšem ještě nejsme zcela spokojeni. Poslední věc k vyřešení je otázka konstrukce objektů. V této fázi průzkumu prototypové dědičnosti nové objekty vytváříme pomocí speciální (tzv. *tovární*) funkce. V příkladu uvedeném výše jsou přítomny dvě, **createCommentObject** a **createBetterCommentObject**. Nevýhodou je, že z takové funkce prostým pohledem nepoznáme, jaký má její volání vliv na prototyp výsledného objektu. Také je trochu nešikovné, že obě dvě vzorové tovární funkce dělají skoro to samé – tedy že v **createBetterCommentObject** nemůžeme zavolat **createCommentObject**. Proto si představíme poslední součást systému prototypové dědičnosti – klíčové slovo **new**.

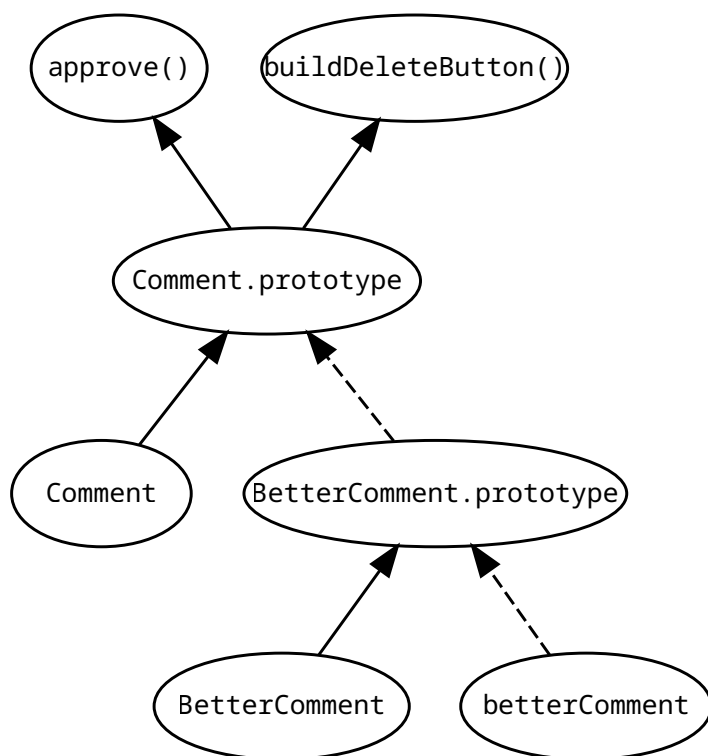
Zápisem **new X()** dojde ke třem hlavním krokům:

1. vznikne nový objekt
2. tento objekt má prototypovou vazbu vedoucí do **X.prototype**
3. funkce **X** je zavolána s **this** nastaveným na nově vzniklý objekt

Vlastnost **prototype** je specifická pro JavaScriptové funkce (každá funkce ji má definovanou ve chvíli svého vzniku). Slouží k jedinému účelu: kdyby se někdo pokusil funkci vykonat pomocí operátoru **new** (tehdy by se použila jako prototypový vzor, viz popis výše). Funkce, které vytváříme za tímto účelem, často pojmenováváme s prvním velkým písmenem a říkáme jim **konstruktory**. Tímto způsobem teď můžeme vylepšit existující prototypový kód:

```
function Comment(data) {
  let node = document.createElement("li");
  this.node = node;
  this.id = data.id;
}
```

```
Comment.prototype.approve = function() {  
    // ...  
}  
  
Comment.prototype.buildDeleteButton = function() {  
    // ...  
}  
  
function BetterComment(data) {  
    Comment.call(this, data);  
}  
  
BetterComment.prototype = Object.create(Comment.prototype);  
  
BetterComment.prototype.checkSpelling = function() {  
    // ...  
}  
  
let betterComment = new BetterComment(data);
```



Obrázek: Prototypová vazba vs. vlastnost prototype

Na obrázku vidíme prototypovou dědičnost v plné parádě. Zápis `Comment.call` v konstruktoru `BetterComment` je nezbytný, protože tím funkci `Comment` nastavujeme správně hodnotu `this` (kdybychom napsali jen `Comment(data)`, hodnota `this` by byla `undefined`).

V moderním JavaScriptovém kódu se ovšem s takovouto definicí objektů a konstruktorů nesetkáme. Ukázalo se, že jejich syntaxe a fungování jsou pro vývojáře často matoucí. Proto se od roku 2015 objevila možnost zápisu tříd klíčovým slovem `class` s funkcionalitou, která je shodná s prototypovou dědičností. To proto, že třídy reprezentují ten samý mechanismus prototypové dědičnosti, jen pro ni používají jinou syntaxi. Skutečně, naše třída `Comment` z vzorového řešení obsahuje vlastnost `prototype` a v ní jsou umístěny všechny metody, které jsme definovali uvnitř bloku `class Comment`. Akorát jsme k tomu nepotřebovali zdlouhavý a hůře pochopitelný zápis `Comment.prototype.x =`

Získané vědomosti o prototypové dědičnosti se nám nicméně mohou hodit, i když pro definici objektů upřednostníme třídy. Vzpomeňme si na to, že díky prototypové vazbě můžeme měnit (či vylepšovat) chování objektů i poté, co byly vytvořeny – tím, že měníme obsah jejich prototypu. To platí nejen pro naše vlastní objekty, ale i pro běžné datové typy. Každý řetězec je totiž instancí funkce **String** (tj. má prototypovou vazbu do **String.prototype**), stejně tak každé pole je instancí funkce **Array**. To nám dovoluje obohacovat chování těchto datových typů. Představme si například, že bychom chtěli mít možnost výběru náhodného prvku z pole. V JavaScriptu je ovšem jen funkce **Math.random()**, která vrací desetinné číslo větší nebo rovno nule a menší než jedna. To je dobrý základ:

```
let jmena = ["Jiří", "Eva", "Petr", "Marie"];

Array.prototype.random = function() {
  let index = Math.floor(Math.random() * this.length);
  return this[index];
}

let jmeno = jmena.random(); // ?
```

Přidáním do objektu **Array.prototype** jsme každé pole *naučili* novému chování. Jen pozor – takovou úpravou vestavěných objektů se vystavujeme riziku kolize. Mohlo by se stát, že v budoucnu v rámci rozšiřování jazyka vznikne oficiální metoda **random**, která by se od té naší mohla lišit. Na druhou stranu je ale vylepšování existujících prototypů skvělým způsobem, jak naše proměnné naučit standardizovanou funkcionalitu, která je příliš nová a v prohlížeči ještě neimplementovaná. Více si o tom povíme v desáté kapitole.

Web Components

Úloha

V šesté kapitole jsme vytvořili single-page aplikaci, která zobrazuje výsledky hledání bez opakovaného načítání stránky. Upravte tento kód opět dle pravidel objektově orientovaného programování, ale tentokrát pomocí konceptu Web Components.

Řešení

Označení Web Components používáme tam, kde se rozhodneme naše třídy definovat jako potomky existujících HTML značek. Znamená to, že takto dokážeme vytvářet vlastní HTML značky, kterým pomocí JavaScriptu dodáme specifickou funkcionalitu (a pomocí CSS specifický vzhled). Myšlenka Web Components je značně obsáhlá a vydala by na samostatnou knihu; vzorové řešení této kapitoly je pro nás proto jen drobnou ochutnávkou této techniky.

HTML, které potřebujeme vytvořit, sestává ze tří částí: hledacího formuláře, prostoru pro výsledky a jednotlivých položek nalezených písní. My zkusíme vytvořit vlastní HTML značku pro formulář (ta bude obsahovat logiku související s HTTP požadavky) a dále pro každý výsledek hledání.

```
<!-- kapitola-9.html -->
<link rel="stylesheet" href="song-result.css" />

<h1>Hledání</h1>
<song-search></song-search>

<script src="song-search.js"></script>
<script src="song-result.js"></script>
```

```
// song-search.js
class SongSearch extends HTMLElement {
```



```

connectedCallback() {
  this.innerHTML = HTML;
  this.querySelector("form").addEventListener("submit", this);
}

async handleEvent(e) {
  e.preventDefault();
  let query = this.querySelector("[name=query]").value;
  let url = `/search?query=${encodeURIComponent(query)}`;
  let response = await fetch(url);
  let songs = await response.json();
  this.showResults(songs, query);
}

showResults(songs, query) {
  let parent = this.querySelector(".results");
  if (songs.length == 0) {
    parent.replaceChildren("Dotazu nevyhovují žádné písně 😞");
    return;
  }

  let heading = document.createElement("h2");
  heading.textContent = `Nalezené písně pro dotaz: ${query}`;


  let ol = document.createElement("ol");
  parent.replaceChildren(heading, ol);

  let results = songs.map(item => {
    let result = document.createElement("song-result");
    result.setData(item);
    return result;
  });
  ol.replaceChildren(...results);
}
}

const HTML = `<form>

```

```

<label>
  Hledaný výraz: <input type="text" name="query" />
</label>
<label><button>

```

```

// song-result.js
class SongResult extends HTMLElement {
  setData(song) {
    this.innerHTML = `
      <a href="${song.url}">${song.title}</a>
      <br/> ${song.text}
    `;
  }
}

customElements.define("song-result", SongResult);

```

```

/* song-result.css */
song-result {
  display: list-item;
}

```

Definice vlastních HTML značek se příliš neliší od běžných tříd, se kterými jsme se seznámili v předchozí kapitole. Aby se s naší třídou dalo pracovat jako s HTML značkou, musíme vykonat tyto dva povinné kroky:

1. Definovat svou třídu jako potomka některé HTML značky. Typicky toho docílíme zápisem **extends HTMLElement**, tedy děděním z obecné HTML značky.
2. Zaregistrovat vzniklou třídu jako definici pro HTML parser. V tomto kroku také dodáme název HTML značky, kterou chceme používat. K tomu slouží globální objekt **customElements** a jeho metoda **define**, pomocí které spárujeme třídu a název. Za zmínku stojí, že v názvu musí být obsažena pomlčka. To garantuje tzv. *dopřednou kompatibilitu*: protože standardní HTML značky ve

svém názvu nikdy pomlčku neobsahují, nestane se v budoucnu, že by došlo ke vzniku nového HTML prvku, který by názvem kolidoval s tím naším vlastním.

Třída **SongResult** je malinká a dovede jedinou věc: převést data jednoho výsledku hledání na HTML. K této nové HTML značce rovnou dodáme také definici stylu, ve které zápisem **display: list-item** zařídíme, aby se naše **<song-result>** korektně zobrazila jako položka seznamu. To proto, že ji máme v plánu umístit do odrážkového seznamu vytvořeného v **<song-search>**.

Optikou objektově orientovaného programování je nezvyklé, že potřebná data nepředáváme konstruktoru třídy **SongResult** a namísto toho tak činíme pomocí metody **setData**. Důvod je ten, že tvorba instancí **SongResult** probíhá zápisem **document.createElement("song-result")** uvnitř třídy **SongSearch**, při kterém nelze žádná data předávat.

Tím se dostáváme k objemnější komponentě **<song-search>**. Většina její implementace je převzata z šesté kapitoly a neobsahuje nic nového. Ani zde nevidíme konstruktor; namísto toho jsme inicializaci (tvorbu formuláře a přidání posluchače) odložili do metody **connectedCallback**. Ta patří k několika tzv. *lifecycle callbackům*; funkcím, které volá prohlížeč, když HTML prvek vkládá do stromu stránky, mění jeho atributy či jej odebírá. Je dobrým zvykem vlastní HTML značky takto inicializovat právě v okamžiku jejich připnutí do dokumentu.

Výsledné HTML je pak triviální a hezky ukazuje, jak jsme funkcionalitu hledacího formuláře skryli, resp. zapouzdřili do implementace komponenty **SongSearch**.

Děděním z třídy **HTMLElement** jsme získali praktické schopnosti rozhraní DOM (**this.innerHTML**, **this.querySelector**). Taktéž jsme získali možnost na naší vlastní značce poslouchat a zejména vytvářet vlastní události – více si o tom povíme v podkapitole pro profíky.

Co jsme se naučili

Po vyřešení osmé úlohy by měl čtenář chápat a ovládat:

- definici vlastní HTML značky rozšířením třídy **HTMLElement**
- registraci názvu vlastní HTML značky

Zelenáči: gettery a settery

Třída **SongResult** pro své fungování potřebuje data, která jí předáváme dedikovanou metodou **setData** (a nikoliv v konstruktoru, protože ten je zpravidla volán bez parametrů). Při objektově orientovaném programování se často setkáváme s podobnými metodami, jejichž účelem je poskytovat či nastavovat data, se kterými objekt pracuje. Vhodné pojmenování v češtině nenalezneme, takže jim říkáme anglicky **getter** a **setter**. V JavaScriptu existuje speciální syntaxe, pomocí které můžeme pro každou vlastnost ve třídě (či obecném objektu) nadefinovat její vlastní getter a setter – funkci volanou při čtení a zápisu hodnoty dané vlastnosti.

Naši metodu **setData** bychom mohli převést na setter takto:

```
class SongResult extends HTMLElement {
  set data(song) {
    this.innerHTML = `
      <a href="${song.url}">${song.title}</a>
      <br/> ${song.text}
    `;
  }
}
```

Všimněme si klíčového slova **set** uvedeného před samotnou metodou. Jeho přítomnost říká, že tato metoda bude volána při každém přiřazení do vlastnosti **data**. S vzniklým HTML prvkem pak budeme pracovat následovně:

```
let songResult = document.createElement("song-result");
songResult.data = data; // z odpovědi HTTP požadavku
```

Při čtení výše uvedeného kódu není na první pohled patrné, že pouhým přiřazením vykonáme nějakou logiku. To je cílem setterů: skrýt před okolním světem skutečnost, že nastavení vlastnosti **data** způsobí další kroky, jako např. vykreslení dalšího HTML ve stránce.

Symetricky je možné definovat i getter. Bývá to obvyklé, i když nikoliv nezbytné. Naše třída **SongResult** ve své současné podobě getter pro vlastnost **data** mít ani nemůže, neboť si získaná data nepamatuje (a proto je nemůže vrátit). Museli bychom ji to naučit takto:

```

class SongResult extends HTMLElement {
  get data() { return this._song; }

  set data(song) {
    this._song = song;
    this.innerHTML = `
      <a href="${song.url}">${song.title}</a>
      <br/> ${song.text}
    `;
  }
}

let songResult = document.createElement("song-result");
songResult.data = data;          // setter
console.log(songResult.data);    // getter

```

SongResult má nyní pro vlastnost **data** getter i setter. Objekt s výsledkem hledání ukládáme do vlastnosti **_song**. Podtržítka na začátku nemá žádný speciální význam, ale v praxi jeho použitím čtenáři naznačujeme, že tato vlastnost je implementační detail třídy a zvenčí by k ní neměl přistupovat (právě proto jsme mu za tímto účelem naimplementovali getter). Je to tedy jakási alternativa *privátních vlastností*, které se objevují v jiných jazycích. V JavaScriptu privátní vlastnosti tříd také existují (stačí, aby jejich název začínal znakem mřížky #), ale nejsou tak populární, protože je nelze použít v rámci dědičnosti.

Zvědavé čtenáře možná napadne, zdali bychom si předaná data mohli uložit do **this.data**? Kód by vypadal takto:

```

class SongResult extends HTMLElement {
  set data(song) {
    this.data = song;
  }
}

```

Takový zápis ovšem nedává smysl, respektive vede na tzv. *nekonečnou rekurzi*: v rámci přiřazení **this.data** je opět volán setter, takže skončíme v nekonečné smyčce a následně narazíme na výjimku způsobenou příliš hlubokým zanořením zásobníku volání.

Koumáci: předávání hodnotou a odkazem

Při předávání dat třídě se ještě zastavíme u otázky, jaká data se dostanou dovnitř volané funkce (ať už je to setter z předchozí podkapitoly, nebo běžná metoda `setData` z původního řešení). Budou to jistě ta samá data, která jsme dostali od serveru. Dostane ale objekt `SongResult` jejich kopii, nebo je bude sdílet s objektem `SongSearch`? A lze toto chování nějak ovlivnit?

Odpověď na tyto otázky je přímočará:

1. Primitivní datové typy (čísla, řetězce, pravdivostní hodnoty, undefined, null) jsou **předávány hodnotou**. Do funkce se dostane kopie hodnoty, resp. uvnitř funkce následně nedokážeme ovlivnit hodnotu mimo funkci.
2. Složité datové typy (zejména objekty, pole, funkce) jsou **předávány odkazem**. Parametr ve funkci nabývá té samé hodnoty jako proměnná předaná při volání (můžeme si jej představit jako odkaz, referenci, ukazatel... podle toho, jaká terminologie nám vyhovuje). Pokud pak ve funkci nějak předaný parametr upravíme (změníme jeho vlastnost, přidáme novou), projeví se to i vně funkce.

Tato pravidla jsou pevná a nelze je upravit. Zejména druhý bod stojí za povšimnutí. Díky tomuto způsobu předávání šetříme paměť (i když proměnná obsahuje velké množství dat, při jejím předání do funkce se nemusí nikam kopírovat), ale zároveň se vystavujeme riziku, že předáním dat dojde k jejich změně, aniž bychom to čekali.

Představme si například úlohu, při které dostaneme od serveru pole uživatelů (u každého bude uvedeno jméno a věk). Máme je následně vypsát a taktéž sdílet, kolik let je nejmladšímu z nich:

```
let users = [
  {name:"Eva", age: 30},
  {name:"Jana", age: 50},
  {name:"Mirek", age: 10}
];

let minAge = getMinAge(users);
console.log("Nejnižší věk: ", minAge);
```

```
users.forEach(console.log);
```

Jak naimplementovat funkci **getMinAge**? Nejsnazší řešení je seřadit uživatele dle věku vzestupně, a pak se podívat na prvního z nich. Každé JavaScriptové pole disponuje metodou **sort**, které stačí předat vhodnou **porovnávací funkci** (protože řazení čehokoliv je vlastně jen opakované porovnávání). Porovnávací funkce dostane dva vzorky a jejím úkolem je vrátit číselnou hodnotu, která vyjadřuje jejich vzájemný poměr:

- nulu, když jsou stejné
- kladné číslo, když je první vzorek větší
- záporné číslo, když je první vzorek menší

V našem případě pak implementace **getMinAge** může vypadat takto:

```
function compareUsers(u1, u2) {
  return u1.age - u2.age;
}

function getMinAge(users) {
  users.sort(compareUsers);
  return users[0].age;
}
```

Jenže pozor! Metoda **sort** pole pozměnila (seřadila). Stalo se tak uvnitř funkce **getMinAge**, nicméně parametr **users** je složitý datový typ, takže změny na něm prováděné uvnitř **getMinAge** se projeví i mimo funkci. Jakmile pak uživatele (globální proměnnou **users**) vypíšeme cyklem **forEach**, zjistíme, že je vypisujeme seřazené – což v zadání rozhodně nebylo.

Co s tím? Nechtěné změně můžeme předcházet na straně volajícího (ten, kdo chce zavolat **getMinAge**, ji musí předat nějaká data, u kterých nevadí změna) i na straně volaného (funkce **getMinAge** se zaváže, že předaná data nebude měnit). V praxi bývá zvykem druhý přístup, neboť v naprosté většině případů se od

funkcí neočekává, že by měnily data, se kterými mají pracovat. Je tedy nutné upravit `getMinAge`, aby se chovala zodpovědněji. Toho můžeme docílit dvěma způsoby:

1. Univerzální řešení je, aby si `getMinAge` před seřazením vytvořila duplikát dat. Toho nedocílíme prostým přiřazením (bystrý čtenář uhodne, proč je hypotetické řešení `let users2 = users; users2.sort()` k ničemu), takže je nutné použít některý z dostupných mechanismů na klonování. V úvahu připadá relativně moderní funkce `structuredClone` (ta vytváří hluboké kopie běžných datových struktur), ale v tomto případě si vystačíme i s jednodušším řešením. Proměnná `users` je pole objektů, takže můžeme snadno vytvořit jeho kopii metodou `slice`, která vrací podmnožinu. Bez zadání parametrů (odkud-kam) vytvoří duplikát:

```
let users2 = users.slice()
```

2. Alternativně můžeme hledat takovou metodu na seřazení pole, která zadany parametr nezmění, ale vrátí nové (seřazené) pole. Taková se do JavaScriptu dostala jako žhavá novinka v roce 2023 a lze ji volat takto:

```
let sortedUsers = users.toSorted(compareUsers);
```

Z příkladu v této podkapitole si bereme ponaučení, že bychom měli naše funkce vést k zodpovědnému chování, pokud jako parametry dostávají složité datové typy. Volající bude v naprosté většině případů očekávat, že jeho data žádným způsobem nepozměníme.

Profíci: vlastní události

Použitím techniky *Custom Elements* vytváříme takové třídy, které dovedou zpracovávat naše data a přitom zůstávají běžnými obyvateli HTML dokumentu. Děděním z `HTMLElement` dostávají vlastnosti a metody, které jsme doposud vídali jen u vestavěných HTML značek. Metodu `addEventListener` nemusíme představovat, ale pro plnohodnotné využití systému událostí si ukážeme druhou stranu této mince: možnost události vytvářet a vyvolávat.

Zatím taková potřeba nebyla, neboť téměř veškeré události v knize probírané vznikaly na základě uživatelské interakce, tj. za jejich vytvoření byl zodpovědný prohlížeč. Pokud ale přistoupíme na skladbu aplikace, při které jsou jednotlivé komponenty realizovány HTML značkami, můžou se nám události hodit jako notificační mechanismus.

Na událost můžeme nahlížet jako na zprávu, která má právě jednoho odesílatele (HTML prvek, na kterém vznikla) a libovolné množství čtenářů (posluchačů). Odesílatel přitom o čtenářích neví, nestará se o jejich počet ani existenci. Je to velmi blízké návrhovému vzoru *PubSub* (*Publisher-Subscriber*). V minulé kapitole jsme uvažovali, jak by moderovaný diskuzní příspěvek mohl dát svému okolí najevo, že je nutné příspěvky znovu načíst – vlastní události by posloužily jako funkční řešení.

V této podkapitole máme třídu **SongResult**, která reprezentuje výsledek hledání. Představme si, že bychom chtěli dát uživateli možnost zpětné vazby, při které by jednotlivé výsledky mohl označovat jako *dobré* a *špatné*. Tuto informaci bychom prostřednictvím HTTP požadavku předali serveru (aby mohl zlepšovat své vyhledávací algoritmy) a zároveň bychom špatné výsledky rovnou nahradili nějakými dalšími.

Takové hodnocení výsledků bude jistě iniciováno uživatelskou interakcí, tj. kliknutím na tlačítko. Která z našich dvou tříd by měla realizovat takový posluchač? Argumenty máme pro obě dvě:

- Poslouchat by měla třída **SongResult**, neboť po kliknutí budeme muset server informovat o uživatelském hodnocení tohoto konkrétního výsledku.
- Poslouchat by měla třída **SongSearch**, neboť pokud půjde o označení *špatného* výsledku, bude pak muset vyvolat nové hledání.

Klíčový argument je ovšem ve prospěch třídy **SongResult**. Skladbu HTML výsledku hledání jsme *zapouzdřili* do této třídy jako její vlastní implementační detail a ostatní objekty o její vnitřní struktuře nemají ponětí. Tlačítko a jeho události si proto bude spravovat výhradně majitel, tedy třída **SongResult**. Abychom pak dali rodičovské třídě **SongSearch** vědět o nutnosti doplnění dalších výsledků, necháme **SongResult** prostě vygenerovat vlastní událost, jakmile dokončíme komunikaci se serverem. Začneme obohacením setteru z první podkapitoly:

```

class SongResult extends HTMLElement {
  set data(song) {
    this._song = song;
    this.innerHTML = `
      <a href="${song.url}">${song.title}</a>
      <br/> ${song.text}
    `;

    let ok = document.createElement("button");
    ok.textContent = "👍";
    ok.addEventListener("click", () => this.sendRating("ok"));

    let ko = document.createElement("button");
    ko.textContent = "👎";
    ko.addEventListener("click", () => this.sendRating("ko"));

    this.append(ok, ko);
  }
}

```

Samotná implementace hodnocení není zajímavá. Bude se jistě jednat o asynchronní funkci, neboť potřebuje komunikovat se serverem. Nás ale zajímá jen ta část, kdy po dokončení komunikace vytvoříme a pošleme vlastní událost:

```

class SongResult extends HTMLElement {
  async sendRating(rating) {
    // await fetch(...)

    let event = new CustomEvent("rating", {
      detail: {rating},
      bubbles: true
    });
    this.dispatchEvent(event);
  }
}

```

Třída **CustomEvent** vytvoří objekt události, který jsme navyklí přijímat jako parametr posluchače. První parametr **CustomEvent** je název události, druhý je volitelný konfigurační objekt. Z jeho klíčů je pro nás zajímavá hodnota **bubbles** (zdali událost bublá, nebo je na ostatních HTML prvcích dostupná jen prostřednictvím zachytávání) a pak **detail**, do kterého můžeme vložit libovolná data. Činíme tak proto, aby posluchač události poznal, k jakému hodnocení došlo. Nakonec událost vyvoláme metodou **dispatchEvent**, což způsobí volání dříve přidáných posluchačů.

Ve třídě **SongSearch** pak budeme tuto událost poslouchat. Posluchače na ni bychom mohli přidávat individuálně na každý vzniklý **<song-result>** (pak by

událost nemusela bublat), ale stejně dobře jej můžeme přidat například na prvek **.results**, nebo přímo na celý **<song-search>**. V takové situaci stačí posluchač přidat jen jednou, nezávisle na tom, kdy a kolik výsledků zobrazujeme:

```
class SongSearch extends HTMLElement {
  connectedCallback() {
    this.innerHTML = HTML;
    this.querySelector("form").addEventListener("submit", this);
    this.addEventListener("rating", this);
  }

  handleEvent(e) {
    switch (e.type) {
      case "submit":
        // existující kód související s provedením hledání
        break;

      case "rating":
        if (e.detail.rating == "ko") {
          // donáčení nových výsledků
        }
        break;
    }
  }
}
```

Práce s vlastními událostmi nabízí ideální mechanismus pro podobné notifikace o komponentách napříč naší aplikací. Využíváme k tomu přitom infrastrukturu, kterou už velmi dobře známe z událostí typu **click** a podobně. Pro úplnost ještě dodejme, že pokud bychom chtěli dát našim třídám možnost pracovat s událostmi (to znamená metody **addEventListener** a **dispatchEvent**), nemusíme je kvůli tomu nutně definovat jako Custom Elements (tedy potomky **HTMLElement**). Stačí, aby byly potomkem jednodušší třídy **EventTarget**. To je k práci s událostmi dostačující, i když tím přijdeme o možnost propagace událostí stromem stránky – toto chování je dostupné jen HTML značkám.

Intl, Storage, polyfilly a další API

Úloha

V rámci webu věnovaného písničkám Karla Gotta připravujeme část s e-shopem. Na prodej budou originály textů jeho písniček s podpisem textaře. Vypište tyto položky; každá bude obsahovat název písničky, datum jejího vzniku a cenu v korunách.

Řešení

Soupis písniček bude opět pocházet z backendu, jehož rozhraní není pro tuto úlohu podstatné. Zajímá nás jen tvar dat jednotlivých položek, které budeme vykreslovat.

Název (řetězec) a cena (číslo) nejsou příliš zajímavé. Datum již představuje jistou výzvu, neboť jeho reprezentace napříč různými informačními systémy bývá různorodá. Zejména v případě formátu JSON (pracujeme s ním pravidelně od páté kapitoly) je nutné formát data s autory backendové komponenty dobře domluvit, protože JSON žádný datový typ pro datum neobsahuje. Možností je celá řada; nejčastěji se v praxi setkáváme se dvěma:

1. Datum zapsané ve formátu řetězce s pevnou strukturou. Standardní zápis bývá sestupně dle velikosti jednotky, tj. např. **1960-05-24**. Tímto způsobem může řetězec obsáhnout několik hodnot naráz (rok, měsíc, den), a pokud by to bylo nutné, můžeme přidat i časovou informaci v rámci dne (hodiny, minuty, ...).
2. Datum zapsané jedním číslem, které vyjadřuje počet časových jednotek uplynulých od vhodně zvoleného počátku. Této formě se říká **timestamp** (časové razítko) a pro správné fungování je nutná dohoda na jednotkách (častá volba jsou sekundy či milisekundy) a počátku (v naprosté většině případů se používá datum 1. 1. 1970). S jediným číslem se pracuje snadno a elegantně, ovšem pro reprezentaci dne v roce to není nejlepší volba. Mimo jiné proto, že pro jedno datum (jeden den) existuje spousta různých hodnot pro timestamp.

Pojďme naši implementaci postavit tak, že vstupní data budou v poli struktur s těmito klíči:

- **id** je unikátní identifikátor písně v databázi (string),
- **name** je název písně (string),
- **price** je cena (číslo),
- **date** je datum ve formátu **rok-měsíc-den** (řetězec), nebo timestamp ve vteřinách (číslo).

```
<!-- kapitola-10.html -->
<h1>Písně k zakoupení</h1>
<ul></ul>

<script src="kapitola-10.js"></script>
```

```
// kapitola-10.js
let DATA = [
  {name: "Kávu si osladím", date: 63111600, price: 123},
  {name: "Lady Carneval", date: "1968-10-07", price: 345},
  {name: "Trezor", date: "1965-04-27", price: 456}
];

let priceOptions = {style: "currency", currency: "CZK"};
let priceFormat = new Intl.NumberFormat(undefined, priceOptions);
let dateOptions = {dateStyle: "long"};
let dateFormat = new Intl.DateTimeFormat(undefined, dateOptions);

function buildItem(item) {
  let li = document.createElement("li");

  let dateIsNumber = (typeof(item.date) == "number");
  let date = (dateIsNumber
    ? new Date(item.date*1000)
    : new Date(item.date)
  );

  li.innerHTML = `
```

```

    <h3></h3>
    <span>Datum: ${dateFormat.format(date)}</span>
    <span>Cena: ${priceFormat.format(item.price)}</span>
    `;
    li.children[0].textContent = item.name; // textContent kvůli XSS
    return li;
}

let items = DATA.map(buildItem);
document.querySelector("ul").replaceChildren(...items);

```

Při přečtení zadání této úlohy nás možná napadne, že bychom k jejímu řešení mohli prostě použít to, co jsme se naučili v minulých kapitolách. Ale pro potřeby výpisu ceny a data by to bylo zbytečně pracné. Je praktičtější se nejprve podívat, jestli nám JavaScript nebo prohlížeč nenabízí nějakou vestavěnou funkcionalitu, díky které bychom mohli ušetřit čas a práci. V tomto případě se jedná o rozhraní **Intl**, jehož název je zkratkou z anglického *internationalization* – funkce související se zobrazováním a zpracováním dat s ohledem na místní zvyklosti. Skutečně, cena i datum jsou příkladem veličin, které se v různých kulturách a jazycích zobrazují různě (a často docela komplikovaně). I kdybychom své stránky cílili výhradně na české uživatele, oceníme, když za nás naše regionální speciality vyřeší někdo jiný.

Globální proměnná **Intl** obsahuje několik tříd určených pro zpracovávání a formátování různých druhů dat, vždy s ohledem na pravidla zadaného jazyka. Ty nejdůležitější jsou:

- **Intl.Collator** pro (abecední) porovnávání řetězců
- **Intl.DateTimeFormat** pro formátování dat a časů
- **Intl.DisplayNames** pro zobrazování názvů zemí, jazyků a měn
- **Intl.ListFormat** pro formátování posloupností hodnot
- **Intl.NumberFormat** pro zobrazování různých číselných hodnot
- **Intl.PluralRules** pro správnou volbu názvu v závislosti na počtu

V naší úloze používáme dva z těchto objektů.

NumberFormat

Pro zobrazení ceny si nejprve nachystáme *formátovací objekt*, tj. instanci třídy **Intl.NumberFormat**. Jejím prvním parametrem je identifikátor jazyka, ve kterém chceme čísla zobrazovat. My předáváme hodnotu **undefined**, která znamená, že prohlížeč má použít svůj výchozí jazyk. To je praktické, neboť uživatelé v různých zemích uvidí naše data vždy dle svých regionálních preferencí.

V druhém parametru (konfiguračním objektu **priceOptions**) uvádíme, jakou číselnou hodnotu zobrazujeme (a v jaké měně). Třída **NumberFormat** dovoluje formátování cen, fyzikálních veličin, procent a obecných čísel. Vzniklý formátovací objekt pak metodou **format** aplikujeme na konkrétní číselná data a zpět dostáváme řetězec určené k zobrazení.

Pro úplnost uveďme několik příkladů tohoto rozhraní:

```
const style = "currency";

new Intl.NumberFormat("cs", {style, currency: "CZK"}).format(1234.56);
// "1 234,65 Kč"

new Intl.NumberFormat("en", {style, currency: "CZK"}).format(1234.65);
// "CZK 1,234.65"

new Intl.NumberFormat("cs", {style, currency: "USD"}).format(123);
// "123,00 US$"

new Intl.NumberFormat("en", {style, currency: "USD"}).format(123);
// "$123.00"
```

Všimněme si například různých oddělovačů tisíců, nebo desetinné tečky (v angličtině) vs. desetinné čárky (v češtině). Implementovat tyto rozdíly ručně by bylo velmi pracné.

DateTimeFormat

S formátovacím objektem typu **DateTimeFormat** se pracuje stejně, jen mu hodnotu musíme předat jako instanci JavaScriptové třídy **Date**. JavaScript totiž má, na rozdíl od JSONu, vestavěný datový typ pro datum a čas. Objekty **Date** reprezentu-

jí bod v čase, a fakticky tak nabízí podobnou funkcionalitu jako zmiňovaný timestamp. V porovnání s číslem ovšem nabízí užitečné metody na čtení a změnu jednotlivých součástí (rok, měsíc, den, hodina, minuta, ...).

Formátování data tím pádem provedeme ve dvou krocích: nejprve ze získané hodnoty (JSON) vytvoříme objekt **Date**, poté provedeme formátování pomocí **Intl.DateTimeFormat**. Víme, že v datech ze serveru může přijít buď řetězec, nebo číslo. Funkce **Date** dokáže přijmout obě tyto hodnoty, ovšem u čísla předpokládá, že timestamp je v milisekundách. Proto příchozí číslo nejprve vynásobíme tisícem.

První parametr pro **DateTimeFormat** je opět identifikátor jazyka, který znovu neuvedeme, aby se použil ten, který je v prohlížeči výchozí. V konfiguračním objektu pak můžeme řadou vlastností upřesnit, které všechny komponenty z data a času chceme vypsát (a s jakou mírou podrobnosti). Nás zajímá jen datum, proto uvádíme klíč **dateStyle** (pro čas bychom přidali ještě **timeStyle**). Ze zvědavosti se podíváme, jaké hodnoty připadají v úvahu:

```
let date = new Date("1982-10-19");

new Intl.DateTimeFormat("cs", {dateStyle:"full"}).format(date);
// "úterý 19. října 1982"

new Intl.DateTimeFormat("cs", {dateStyle:"long"}).format(date);
// "19. října 1982"

new Intl.DateTimeFormat("cs", {dateStyle:"medium"}).format(date);
// "19. 10. 1982"

new Intl.DateTimeFormat("cs", {dateStyle:"short"}).format(date);
// "19.10.82"
```

Co jsme se naučili

Po vyřešení deváté úlohy by měl čtenář chápat a ovládat:

- reprezentaci data v JavaScriptu a JSONu
- smysl rozhraní **Intl**

- práci s objekty `Intl.NumberFormat` a `Intl.DateTimeFormat`

Zelenáči: Web Storage

Protože úloha v této kapitole je motivována e-shopem, mohli bychom si vyzkoušet implementaci jednoduchého nákupního košíku. Do něj smí uživatel vložit položky, o které má zájem, a při jejich výpisu bude tato skutečnost zmíněna. Klíčovým atributem nákupního košíku je, že jeho obsah je dostupný i po znovunačtení stránky. Typicky je toho docíleno ukládáním obsahu košíku na serveru, ale v této knize se soustředíme na klientský JavaScript a zároveň je to skvělý způsob, jak si vyzkoušet další užitečné a přitom velmi snadné rozhraní – **Web Storage**.

Prohlížeč nám prostřednictvím Web Storage nabízí možnost uložit data tak, abychom se k nim dostali při všech dalších návštěvách dané stránky. Přesněji, tato data jsou dostupná *všem stránkám na té doméně, ve které byla uložena*. Máme tak zaručeno, že se k takto uloženým datům nedostanou skripty vložené do cizích stránek (tj. stránek na jiných doménách).

S rozhraním Web Storage pracujeme prostřednictvím globální proměnné **localStorage**, která nabízí dvě hlavní metody:

```
// zápis
localStorage.setItem(key, value);

// čtení
let value = localStorage.getItem(key);
```

Vidíme, že do Web Storage ukládáme dvojice klíč-hodnota. Jak klíče, tak hodnoty musí být obyčejné řetězce. Pokud tedy chceme uložit složitější data (jako například nákupní košík), budeme je muset pro potřeby uložení převést na řetězec.

Nákupní košík můžeme reprezentovat různými způsoby, nejjednodušeji jako pole identifikátorů těch položek, které jsou v košíku. Abychom toto pole mohli vložit do **localStorage**, můžeme jej převést na řetězec například metodou

JSON.stringify (dostaneme řetězec ve formátu JSON). Při načtení stránky pak košík z minula, pokud nějaký je, získáme zpět z **localStorage** a převedeme na pole metodou **JSON.parse**:

```
let storedData = localStorage.getItem("shopping-cart");
let shoppingCart = storedData ? JSON.parse(storedData) : [];

function addToCart(id) {
  shoppingCart.push(id);
  let data = JSON.stringify(shoppingCart);
  localStorage.setItem("shopping-cart", data);
}

function removeFromCart(id) {
  let index = shoppingCart.indexOf(id);
  if (index !== -1) {
    shoppingCart.splice(index, 1);
    let data = JSON.stringify(shoppingCart);
    localStorage.setItem("shopping-cart", data);
  }
}
```

Odebírání položky z pole je komplikovanější. Nejprve musíme zjistit její *index* (tj. na kolikátém místě se nachází), a pak z pole odstranit prvek dle indexu. Používáme k tomu metodu **indexOf**; ta pro neexistující prvek vrátí speciální hodnotu **-1**.

Jistou alternativou by bylo ukládání položek v množině (objekt **Set**), u které je – na rozdíl od pole – zaručena jedinečnost, a proto je možné prvek odebrat bez znalosti indexu. Množinu ale neumíme přímočaře reprezentovat ve formátu JSON, takže se budeme držet obyčejného pole. Pak už jen stačí ke každé položce přidat tlačítko:

```
function buildCartButton(item) {
  let button = document.createElement("button");

  if (shoppingCart.includes(item.id)) {
    button.textContent = "Odebrat z košíku";
  }
}
```

```

} else {
    button.textContent = "Přidat do košíku";
}

button.addEventListener("click", () => {
    if (shoppingCart.includes(item.id)) {
        removeFromCart(item.id);
    } else {
        addToCart(item.id);
    }

    let newButton = buildCartButton(item);
    button.replaceWith(newButton);
});

return button;
}

```

Jakmile dojde ke změně obsahu košíku, musíme tuto skutečnost zohlednit v uživatelském rozhraní. Jednou z možností by byla změna textu na existujícím tlačítku. Tím bychom ale měli kód s nastavováním textu dvakrát. Proto je snazší vyrobit nové tlačítko (se správným novým textem) a nahradit jím to původní.

Koumáci: Polyfilly

Při tvorbě webových aplikací často narážíme na otázku kompatibility napříč prohlížeči. Jejich nové verze vznikají s vysokou frekvencí a může se stát, že při psaní JavaScriptového kódu použijeme jinou verzi, než jakou pak bude mít uživatel při prohlížení našeho webu. Dopředná kompatibilita (použití staré funkcionality v modernějších verzích) typicky nebývá problém, neboť webové standardy – HTML, CSS, JavaScript – se snaží o zachování maximální podpory všech historicky existujících funkcí. Horší ale je, když například s novou verzí standardu HTML či ECMAScript přibude užitečné API, které bychom rádi použili, ale musíme počítat i s těmi uživateli, ke kterým se pokročilá implementace zatím nedostala.

Příkladem může být právě některý objekt ze standardu **Intl**. Pokud bychom neměli jistotu, že každý uživatel má prohlížeč s třídou **Intl.DateTimeFormat**, jak bychom při řešení úlohy z této kapitoly postupovali?

Konzervativní přístup je formátovat datum bez použití **Intl**. Víme, že je to možné, ale zároveň je to pracné a málo flexibilní (s ohledem na všechny možné jazyky našich uživatelů). Mohli bychom se proto uchýlit k podmínce:

```
let date = new Date();

if (Intl.DateTimeFormat) {
  // automatické, chytré řešení
  let options = {dateStyle:"long"};
  let dateFormat = new Intl.DateTimeFormat(undefined, options);
  console.log(dateFormat.format(date));
} else {
  // ruční, slabé řešení
  let parts = [
    date.getDate(),
    date.getMonth()+1,
    date.getFullYear()
  ];
  let str = parts.join(". ");
  console.log(str);
}
```

Nevýhodou je, že náš kód je nyní zbytečně *chytrý* – obsahuje dvě různé varianty formátování data. Pokud bychom datum zpracovávali na více místech, tato neoptimalita by byla ještě patrnější. Podíváme se proto na alternativní přístup, kterému se říká **polyfill**.

Koncept polyfillu (český ekvivalent neexistuje) je specifikum JavaScriptu; v jiných jazycích se s ním nesetkáváme. Jeho podstatou je skutečnost, že JavaScript je velmi dynamický, a pokud v něm nějaká globálně dostupná funkcionalita schází, můžeme ji za jistých podmínek prostě doplnit. V praxi to znamená, že do stránky vložíme dva skripty: nejprve soubor s polyfillem, který zmiňovanou funkci implementuje; poté naši aplikaci, která se už nemusí rozhodovat podle dostupnosti, neboť díky polyfillu je potřebná funkce vždy dostupná.

Zkusme si jednoduchou verzi polyfillu pro **DateTimeFormat** naimplementovat sami:

```
class DateTimeFormat {
  constructor(language, options) {
    this.language = language; // neumíme zohlednit
    this.options = options;
  }

  format(date) {
    // výhledově bychom měli zohlednit this.options.dateStyle
    let parts = [
      date.getDate(),
      date.getMonth()+1,
      date.getFullYear()
    ];
    return parts.join(". ");
  }
}

if (!Intl.DateTimeFormat) {
  Intl.DateTimeFormat = DateTimeFormat;
}
```

Nedílnou součástí každého dobrého polyfillu je **feature testing**: musíme ověřit, zdali prohlížeč námi dodávanou funkcionalitu neumí sám od sebe. Pokud ano, bylo by zbytečné (v tomto případě dokonce škodlivé) dodávat tu naši. Proto do **Intl.DateTimeFormat** přiřazujeme v podmínce na konci polyfillu.

Výše uvedený polyfill v této formě nelze považovat za plnohodnotnou náhradu objektu **DateTimeFormat**. Jednak nerespektuje zadaný jazyk, jednak vůbec není možné konfigurovat formát výpisu pomocí druhého parametru konstruktoru. Zvědavý čtenář může ale snadno metodu **format** obohatit i o další formáty data či času.

Mimochodem: pokud bychom narazili na opravdu starý prohlížeč, mohlo by se stát, že by v něm vůbec nebyla globální proměnná **Intl**. Pak bychom museli kód na konci polyfillu upravit následovně:

```

if (!window.Intl) { window.Intl = {}; }

if (!Intl.DateTimeFormat) {
  Intl.DateTimeFormat = DateTimeFormat;
}

```

V praxi se s polyfilly setkáváme celkem často. To proto, že řada JavaScriptových funkcí vzniká za účelem zvýšení vývojářova komfortu, ale koncepčně nic nového nepřináší. V aplikacích pak můžeme vidět plnohodnotné polyfilly například pro funkci **fetch**, objekt **Promise** nebo třeba **String.prototype.padStart**.

Ne vždy ovšem můžeme polyfill použít. V těchto dvou případech máme smůlu:

1. Když se jedná o **změnu syntaxe** jazyka (příklad: klíčová slova **async/await**, **arrow** funkce, **class** a další). Kód našeho polyfillu nedovede naučit parser JavaScriptu novým pravidlům; jeho prostřednictvím můžeme jen přidávat nové vlastnosti a funkce existujícím rozhraním.
2. Když potřebná funkcionalita není v prohlížeči žádným způsobem dostupná (příklad: Web Storage, **XMLHttpRequest**, další API z následující podkapitoly). V takové situaci sice umíme pomocí polyfillu dodat potřebné rozhraní, ale neumíme zařídit jeho (dostatečnou) funkčnost.

Studium či dokonce vlastní tvorba polyfillů je každopádně skvělý způsob, jak si osvojit různá JavaScriptová rozhraní a seznámit se s jejich fungováním. Pojdme si na závěr této podkapitoly vyzkoušet polyfill (tentokrát plně funkční) pro funkcionální iteraci, konkrétně například pro metodu **Array.prototype.map**:

```

if (!Array.prototype.map) {
  Array.prototype.map = function(callback, thisArg) {
    let results = [];
    let arr = this;
    for (let i=0; i<arr.length; i++) {
      let result = callback.call(thisArg, arr[i], i, arr);
      results.push(result);
    }
    return results;
  }
}

```

Profíci: další API

V této kapitole jsme si ukázali, že v prostředí (nejen) klientského JavaScriptu je k dispozici řada rozhraní, která nám mohou významně usnadnit práci. Jsou to tzv. *domain-specific* APIs; nástroje, které používáme situačně. Někdy je nepotřebujeme vůbec, jindy jsme za jejich přítomnost velmi vděční. Podobných rozhraní jsou k dispozici desítky a jejich popis by vystačil na celou další knihu. Pojďme si v podkapitole pro profíky alespoň v rychlosti popsat ta, která jsou nejzajímavější a nejužitečnější.

- **Web Crypto:** funkce pro generování náhodných čísel, šifrovacích klíčů, hashování, šifrování a dešifrování, podepisování a ověřování.
- **Web Audio:** funkce pro generování zvuku, jeho transformaci a analýzu.
- **Web MIDI:** rozhraní pro spolupráci s externími zařízeními, které generují či přijímají tzv. *MIDI události* (používají se zejména v hudební produkci).
- **Gamepad API:** získávání stavu ovládacích prvků na herních ovladačích.
- **Sensor APIs:** široká sada funkcí pro zjišťování stavu různých hardwarových senzorů, kterými může být počítač (častěji telefon) vybaven. To zahrnuje kompas, akcelerometr, gyroskop, čidlo osvětlení, stav baterie a další.
- **Clipboard:** přístup (čtení i zápis) k schránce operačního systému.
- **Web Speech:** převod textu na řeč, rozpoznávání textu.
- **Web Workers:** podpora pro vykonávání JavaScriptového kódu ve více opravdových vláknech.
- **Web Sockets / SSE / WebRTC:** alternativní protokoly pro přenos dat tam, kde HTTP nestačí.
- **Bluetooth / Web Serial:** pro přístup k zařízením připojeným pomocí bluetooth či (virtuálního) sériového portu.

Z výčtu jsou záměrně vypuštěna dvě rozhraní, určená pro práci s grafikou (SVG a Canvas). To proto, že se jim budeme věnovat v dalších kapitolách.

HTML Canvas

Úloha

Rádi bychom uživatelům, kteří se na našem webu zaregistrují, dovolili nahrát profilový obrázek. Tento se bude zobrazovat vedle jejich jména u diskuzních příspěvků a dalších aktivit. Protože se bude zobrazovat malý, není vhodné, aby nám uživatelé nahrávali své portréty ve velkém rozlišení. Proto obrázek před nahráním zmenšíte tak, aby jeho delší strana nepřesahovala zadaný limit (např. 1000 pixelů).

Řešení

```
<!-- kapitola-11.html -->
<input type="file" accept="image/*" />
<script src="kapitola-11.js"></script>
```

```
// kapitola-11.js
const MAX = 1000;

async function loadImage(file) {
  let img = new Image();
  img.src = URL.createObjectURL(file);
  await img.decode();
  return img;
}

function resizeImage(img) {
  const { width, height } = img;
  let scale = Math.max(width/MAX, height/MAX, 1);
  let canvas = document.createElement("canvas");
  canvas.width = Math.round(width / scale);
  canvas.height = Math.round(height / scale);
```

```

    let ctx = canvas.getContext("2d");
    ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
    return canvas.toDataURL("image/jpeg");
}

async function onChange(e) {
    let file = e.target.files[0];
    if (!file) return;

    let sourceImage = await loadImage(file);
    let targetImage = resizeImage(sourceImage);

    fetch("/upload", {
        method: "POST",
        body: targetImage
    });
}

let input = document.querySelector("[type=file]");
input.addEventListener("change", onChange);

```

Přestože vzorové řešení má jen kolem třiceti řádků, obsahuje řadu novinek. Pro snazší pochopení si úlohu rozdělíme na tři menší části: získání obrazových dat z disku uživatele, zmenšení a následné nahrání na server.

Načtení obrázku

V klientském JavaScriptu nelze přistupovat k libovolným souborům na disku uživatele. Představovalo by to značné bezpečnostní riziko, neboť při návštěvě webové stránky by cizí skript mohl snadno číst naše citlivá data a následně je například posílat na server útočníka. Panuje proto pravidlo, že prohlížeč zpřístupní obsah jen takových souborů, které za tímto účelem uživatel explicitně označil. Možnosti jsou dvě: buď takový soubor uživatel myší přetáhl z plochy operačního systému do okna prohlížeče, nebo nějaký soubor vybral prostřednictvím formulářového prvku `<input type=file>`.

Druhá možnost je o něco přímočařejší a také častější, takže ji využijeme pro vzorové řešení. Atributem **accept** omezíme typ použitelných souborů, takže následně nemusíme uvažovat variantu, kdy by uživatel vybral soubor, který vůbec není obrázek. Po vybrání souboru vznikne událost **change**, a tím dojde k vykonání posluchače **onChange**. Vlastnost **files** HTML prvku **<input>** v takové chvíli obsahuje pole vybraných souborů (víc by jich mohlo být, pokud bychom použili atribut **multiple**). Jedná se o proměnné typu **File** – s takovým objektem jsme se zatím nesetkali.

Objekt **File** slouží k přístupu k souboru, ale nedovoluje nám přímo získat jeho obsah. S proměnnou tohoto typu můžeme provádět následující hlavní operace:

1. Můžeme ji předat funkci **fetch**, a tím obsah souboru nahrát. To dělat nechceme, protože bychom nahrávali originální obrázek, který může být příliš velký.
2. Můžeme použít objekt **FileReader**, prostřednictvím kterého bychom se dostali k obsahu souboru (jednotlivým bajtům). To vypadá užitečně, ale jedná se o příliš nízkoúrovňový přístup. Museli bychom v JavaScriptu implementovat dekódování obrazového formátu (JPEG, PNG, GIF, ...), což je zbytečně složité a pracné.
3. Protože víme, že vybraný soubor odpovídá obrázku, můžeme jej využít k vytvoření HTML prvku ****. Ten je vhodný pro následné zmenšování.

HTML značka **** ovšem vyžaduje URL, které nemáme. Naštěstí existuje šikovná a nepříliš známá funkce **URL.createObjectURL**, která slouží právě k tomuto účelu. Pro předaný objekt typu **File** vytvoří speciální dočasné URL, které můžeme až do zavření stránky používat tam, kde je URL očekáváno. V našem případě ho předáme do atributu **src** nově vytvořeného obrázku.

Ještě než začneme tento obrázek zpracovávat, musíme počkat, než jej prohlížeč načte (teprve potom budou například známy jeho rozměry). Načtení obrázku je asynchronní proces a máme dva způsoby, jak s ním pracovat:

1. Počkáme na událost **load**. To by znamenalo přidání posluchače a zabalení následného kódu do vlastní funkce. Bude to fungovat, ale existuje snazší řešení.
2. Použijeme relativně moderní metodu **decode()**, která taktéž čeká na načtení a vrací Promise. To je pro náš případ ideální.

Změna rozměrů

Uživatelé vybraný soubor máme nyní načtený v HTML obrázku. Jeho rozměry můžeme získat dvěma způsoby:

1. Pomocí vlastností **naturalWidth** a **naturalHeight**. Ty obsahují původní rozměr obrázku v pixelech tak, jak jsou obsaženy v souboru.
2. Pomocí vlastností **width** a **height**. Ty odpovídají rozměrům obrázku při jeho vykreslení. V takovou chvíli může být velikost ovlivněna HTML atributy nebo CSS vlastnostmi, proto je tento způsob zjišťování rozměrů méně spolehlivý. Pokud ovšem obrázek není součástí stromu stránky, hodnoty **width** a **height** odpovídají těm původním (a shodují se tedy s výše představenými vlastnostmi **naturalWidth** a **naturalHeight**). To je náš případ, a proto sáhneme po těchto kratších názvech.

Zvědavý čtenář si možná povšiml nezvyklého přístupu k vlastnostem obrázku. Použili jsme syntaxi, která se nazývá **destructuring**:

```
const { width, height } = img;
```

Tento zápis, při kterém se kolem definovaných proměnných píšou složené (nebo hranaté) závorky, odpovídá následujícímu kódu:

```
const width = img.width;
const height = img.height;
```

Destructuring nám dovoluje ušetřit trochu místa ve chvíli, kdy vytváříme nové proměnné ze stejně pojmenovaných vlastností objektu na pravé straně rovnítka. Použít jej můžeme taktéž, pokud do proměnných obdobným způsobem *vybalujeme* položky z pole.

K výpočtu cílových rozměrů si dále nachystáme **měřítko**: číselnou hodnotu, která říká, kolikrát je nutné obrázek zmenšit. Protože chceme zachovat původní poměr stran, budeme oba rozměry zmenšovat stejně. Měřítko definujeme tímto vzorcem:

```
let scale = Math.max(width/MAX, height/MAX, 1);
```

První parametr funkce `max` říká, *kolikrát je šířka obrázku větší, než požadovaný limit*. Druhý parametr funguje shodně, jen pro výšku. Z těchto hodnot bereme maximum, neboť v zadání je požadováno takové zmenšení, po kterém budou obě strany menší nebo rovny limitu.

Jednička na konci je malý trik: mohlo by se stát, že by nám uživatel nahrál obrázek, jehož oba rozměry jsou menší než náš zmenšovací limit. Měřítka by pak bylo menší než jedna a při následné transformaci bychom proto obrázek *zvětšili*. To je zbytečné a nežádoucí, takže pro malé obrázky zvolíme měřítko rovno jedné, a tím pádem zachováme jejich původní velikost.

Nové rozměry pak vypočítáme jako ty původní vydělené měřítkem. V tuto chvíli nám již ovšem nestačí HTML obrázky. I kdybychom jim nastavovali menší rozměry, prohlížeč provede zmenšení jen pro potřeby zobrazení. Nahraná obrazová data budou stále v původním rozlišení. Proto využijeme HTML značku `<canvas>`, která představuje univerzální nástroj pro tvorbu a manipulaci s rastrovými obrazovými daty.

Prvek `<canvas>` (anglické označení pro malířské plátno) je sice součástí jazyka HTML, ale pracujeme s ním výhradně pomocí JavaScriptových funkcí a vlastností. A že jich je! Zatím si ukážeme jen úplný základ práce s canvasem, ale v následujících podkapitolách prozkoumáme i další jeho možnosti.

Naše plátno potřebuje především nastavit rozměry. K tomu použijeme výše zmíněný výpočet:

```
let canvas = document.createElement("canvas");
canvas.width = Math.round(width / scale);
canvas.height = Math.round(height / scale);
```

Canvas odpovídá obdélníku pixelů, proto je potřeba jeho rozměry nastavovat v celých číslech. Měřítka i výsledek dělení mohou být čísla desetinná, takže je na celé pixely musíme zaokrouhlit.

Funkce, které canvas nabízí, jsou v rámci jeho JS API seskupeny do tzv. *kontextů* – JavaScriptových objektů specializovaných pro různé vykreslovací operace. Pro naši úlohu je vhodný kontext nazvaný prostě **2d**. Ten dovoluje takové operace, které známe například z programu Malování ve Windows: tah štětcem, vyplňování plochy, vkládání obrázků a podobně. Čistě teoretickým příkladem dalšího

kontextu může být **webgl**, což je vysoce specializované rozhraní určené pro vykreslování pomocí akcelerované grafiky prostřednictvím jazyka OpenGL. S takto pokročilými technikami se ovšem v této knize nesetkáme.

Metoda **drawImage** patří dvourozměrnému kontextu a má řadu různých signatur. Prvním parametrem je zdrojový obrázek, další dva parametry definují cílové souřadnice (místo, na kterém v canvasu vykreslíme levý horní roh obrázku) a nejdůležitější jsou poslední dva parametry – cílové rozměry. Právě díky nim prohlížeč vykreslí původní obrázek do canvasu zmenšený.

Celá tato zmenšovací operace nás stojí jen dva řádky kódu:

```
let ctx = canvas.getContext("2d");
ctx.drawImage(img, 0, 0, canvas.width, canvas.height);
```

Nahrání obrázku

Máme skoro hotovo! Zmenšili jsme portrét, jen ho teď nemáme v HTML obrázku, ale v HTML canvasu. A ten nelze sám o sobě vzít a nahrát na server. Mimo jiné proto, že canvas představuje dvourozměrné pole pixelů, ale soubory obsahují obrazová data v nějakém formátu. Musíme proto prohlížeč nejprve požádat, aby obrazová data z canvasu vhodně zakódoval.

Canvas pro tyto účely disponuje dvěma metodami, které dělají v podstatě to samé: **toDataURL** a **toBlob**. Liší se takto:

- Metoda **toBlob** je asynchronní (přijímá callback) a produkuje objekt typu **Blob**, který zastřešuje výsledná binární data. Můžeme jej předat funkci **fetch** a odeslat na server. Tato metoda představuje výkonnější řešení: je úsporná a díky asynchronnosti neblokuje hlavní vlákno.
- Metoda **toDataURL** je synchronní a její návratová hodnota je tzv. **data URI**: řetězec ve tvaru podobném URL, obsahující zakódovaná obrazová data. Můžeme jej použít všude, kde je očekáváno URL (podobně jako výsledek volání **URL.createObjectURL**). Takový řetězec můžeme samozřejmě taktéž na-
hrát na server. Kromě blokování hlavního vlákna je další nevýhoda metody

toDataURL v tom, že výsledný řetězec obsahuje jednotlivé bajty zapsané pomocí *kódování Base64*. V něm se používají pouze tišitelné znaky, čímž datový objem naroste zhruba o třetinu.

Pro jednoduchost ve vzorovém řešení použijeme **toDataURL**. Jejím parametrem je identifikace obrazového formátu, do kterého chceme obrázek zakódovat:

```
function resizeImage(img) {
  // tvorba canvasu viz výše
  return canvas.toDataURL("image/jpeg");
}

let targetImage = resizeImage(sourceImage);
```

V porovnání s minulými kapitolami potřebujeme na server odeslat značné množství dat. Musíme proto použít HTTP metodu **POST** a data umístit do těla požadavku. V případě funkce **fetch** je to jen otázka konfiguračních parametrů **method** a **body**:

```
fetch("/upload", {
  method: "POST",
  body: targetImage
});
```

Tím jsme na server poslali data zmenšeného zakódovaného obrázku.

Co jsme se naučili

Po vyřešení desáté úlohy by měl čtenář chápat a ovládat:

- možnosti přístupu k datům uživatelem vybraného souboru
- využití HTML canvasu pro zmenšení obrázku
- získání obrazových dat z canvasu ve formátu vhodném k odeslání po síti

Zelenáči: další dovednosti canvasu

HTML značka `<canvas>` je univerzální kreslicí plocha. Ukázali jsme, jak do ní přenést obrázek ze značky ``; nyní přišel čas vyzkoušet další metody, které 2d-kontext nabízí. Abychom nemalovali jen tak nahodile, vyřešíme skutečnou úlohu, ke které lze canvas použít – ukazatel načítání.

Předpokládejme, že provádíme operaci, která trvá delší dobu. Může jít třeba o nahrávání profilového obrázku nebo čekání na nějaká data ze serveru. Když se jedná o aktivitu na pozadí, uživatel nemá žádné informace o její existenci případně průběhu. Bývá proto obvyklé, že síťovou komunikaci nějakým nenápadným způsobem vizualizujeme. Sestavme animaci, která představuje postupně se zvětšující kruhovou výseč (zelené barvy). Abychom nemuseli vymýšlet, kam takový obrázek umístit, zobrazíme ho v záhlaví záložky prohlížeče vedle titulku stránky – uděláme z tohoto obrázku tzv. **favikonku**.

Favikonky jsou malé obrázky, které k HTML dokumentu připojujeme pomocí značky `<meta>` v hlavičce stránky. Typicky to vypadá takto:

```
<meta rel="icon" href="ikonka.png" />
```

Nic nám ale nebrání takovou favikonku vyrobit pomocí JavaScriptu. Jakmile pro ni dokážeme sestavit URL, můžeme jej použít pro změnu atributu `href` u značky `<link>`.

Začneme tím, že si nachystáme malý čtvercový canvas. Pro rozměry vytvoříme konstantu:

```
const SIZE = 32;
let canvas = document.createElement("canvas");
canvas.width = canvas.height = SIZE;
let ctx = canvas.getContext("2d");
```

Malování obstará funkce, která pomocí metod kontextu `ctx` vytvoří požadovaný útvar. Ta bude muset znát úhel (velikost výseče). Proto si nachystáme kód, který bude v čase úhel měnit a periodicky kreslicí funkci vykonávat:

```
let angle = 0;
setInterval(() => {
  angle = (angle + 0.1) % (2 * Math.PI);
```



```
draw(angle);
}, 100);
```

V počítačové grafice bývají úhly zpravidla specifikovány pomocí radiánů. V tomto kódu každých 100 milisekund zvětšíme úhel o 0,1 radiánu. Hodnotu ještě upravíme operátorem procenta (zbytek po dělení) tak, aby nikdy nepřesáhla 2π , tj. celý kruh. To znamená, že jakmile vykreslíme kruh, začneme znovu od začátku úzkou výsečí od nuly.

Zbývá naimplementovat kreslení ve funkci **draw**. HTML canvas je zajímavý tím, že jakmile do něj nějaká data vykreslíme, už je zpětně nemůžeme měnit. Animaci tedy realizujeme tak, že canvas nejprve zcela vyprázdníme, a pak do něj vykreslíme nový snímek dle aktuálního stavu. Pro vymazání existuje metoda **clearRect**, které předáme souřadnice (a rozměry) obdélníkové oblasti, kterou chceme vymazat. V našem případě je to celý canvas:

```
function draw(angle) {
    ctx.clearRect(0, 0, SIZE, SIZE);
}
```

Kruhovou výseč vytvoříme pomocí operace, která odpovídá tažení virtuálním štětcem po plátně. Sestává z těchto kroků:

1. posun štětce doprostřed plátna
2. čára doprava
3. kruhový oblouk o délce úměrné požadovanému úhlu
4. čára zpět do středu
5. vyplnění vzniklé oblasti barvou

Náš kontext nabízí metody přesně pro tyto kroky:

```
ctx.beginPath();
ctx.moveTo(SIZE/2, SIZE/2);
ctx.lineTo(SIZE, SIZE/2);
ctx.arc(SIZE/2, SIZE/2, SIZE/2, 0, angle);
ctx.closePath();
```

```
ctx.fillStyle = "green";
ctx.fill();
```

Jediné komplikované místo je metoda **arc**, která vytváří kruhový oblouk. Ten vyžaduje mnoho parametrů: dvě souřadnice středu kruhu, poloměr, počáteční úhel (nula odpovídá bodu na kružnici nejvíc vpravo) a cílový úhel.

Za zmínku ještě stojí nastavení barvy výplně. Stejně jako v programu Malování, i zde provádíme nastavení barvy (vlastnost **fillStyle**) nezávisle na následném kreslení (metoda **fill**). Ve chvíli vykreslení se pak použije aktuálně nastavená barva. Znamená to, že změnu barvy můžeme provést kdykoliv před vykreslením, ale nikdy až poté.

V tuto chvíli máme v canvasu hotový obrázek. Zbývá z něj vytvořit URL (to umíme z první části této kapitoly) a nastavit jej prvku **<meta>**:

```
let link = document.head.querySelector("[rel=icon]");
link.href = canvas.toDataURL("image/png");
```

Výsledný kód pak celý vypadá například takto:

```
<!-- kapitola-11-zelenaci.html -->
<!doctype html>
<html>
  <head><link rel="icon" /></head>
  <script>

let canvas = document.createElement("canvas");
const SIZE = 32;
canvas.width = canvas.height = SIZE;
let ctx = canvas.getContext("2d");

function draw(angle) {
  ctx.clearRect(0, 0, SIZE, SIZE);
  ctx.beginPath();
  ctx.moveTo(SIZE/2, SIZE/2);
  ctx.lineTo(SIZE, SIZE/2);
  ctx.arc(SIZE/2, SIZE/2, SIZE/2, 0, angle);
```

```

    ctx.closePath();
    ctx.fillStyle = "green";
    ctx.fill();

    let link = document.head.querySelector("[rel=icon]");
    link.href = canvas.toDataURL("image/png");
}

let angle = 0;
setInterval(() => {
    angle = (angle + 0.1) % (2 * Math.PI);
    draw(angle);
}, 100);

</script>
</html>

```

Koumáci: obrazová data z kamery a videa

Zatím jsme ukázali, jak v canvasu vytvářet nový obsah a jak ho z canvasu získat v podobě URL. Prohlížeče nabízí ještě další způsoby, jak do canvasu dostat obrazová data – ovšem stále prostřednictvím metody **drawImage**, kterou jsme si představili ve vzorovém řešení.

Její první parametr specifikuje zdroj dat. Nemusí jít jen o obrázek; může to být také jiný canvas nebo HTML značka **<video>**. Pokud bychom tedy měli soubor s videem, můžeme jej přehrávat prostřednictvím značky **<video>**, její obsah pravidelně přenášet do canvasu a tam jej dle potřeby zpracovávat. Je to docela snadné:

```

<video src="video.mp4"></video>
<canvas></canvas>

```

```

function drawToCanvas() {
    let video = document.querySelector("video");
    let canvas = document.querySelector("canvas");
    canvas.width = video.videoWidth;

```

```

    canvas.height = video.videoHeight;

    let ctx = canvas.getContext("2d");
    ctx.drawImage(video, 0, 0);

    requestAnimationFrame(drawToCanvas);
}

drawToCanvas();

```

Pro periodické vykonávání funkce **drawToCanvas** jsme tentokrát zvolili jiný přístup než **setInterval**. Pokud chceme opakovaně provádět operaci, která něco vykresluje do stránky, je lepší použít funkci **requestAnimationFrame**. U ní neříkáme časový krok, tj. *za jak dlouho má prohlížeč zadáný kód vykonat*. Namísto toho necháváme prohlížeč, aby volil časový krok sám – s ohledem na aktuální vytížení počítače, snímkovou frekvenci monitoru a další faktory (pokud je například záložka s naším kódem neaktivní, uživatel nic nevidí a prohlížeč může snížit frekvenci vykreslování).

Funkce **requestAnimationFrame** je, podobně jako **setTimeout**, jednorázová. Musíme ji proto zavolat po každém vykreslení (a naplánovat tak zase další krok). Vytváříme tím nepřímo smyčku, kterou můžeme přerušit prostě tím, že přeskočíme další volání **requestAnimationFrame**.

Vykreslování a zpracování jednotlivých snímků z HTML značky **<video>** je zároveň základem pro přístup k datům z kamery. Docílíme toho v několika fázích:

1. požádáme prohlížeč o tzv. **stream** (datový proud) z kamery
2. prohlížeč požádá uživatele o svolení – jedná se o vysoce citlivá data, takže uživatel musí vyjádřit souhlas
3. získaný stream použijeme jako zdroj dat pro značku **<video>**
4. obrazová data jednotlivých snímků získáme do canvasu stejně, jako v minulé ukázce

Přestože se dohromady jedná o spoustu práce, většinu z ní udělá prohlížeč. My musíme v JavaScriptu nově jen vyvolat žádost o přístup ke streamu. K tomu slouží funkce **navigator.mediaDevices.getUserMedia** a její jediný parametr

upřesňuje, která všechna audio-vizuální data z kterých vstupů nás zajímají. Můžeme zde poměrně detailně určit, jestli stojíme jen o obraz, nebo i o zvuk; u obrazu definovat požadované rozlišení a dále třeba prohlížeči naznačit, jestli má v případě mobilního telefonu použít hlavní zadní kameru, nebo selfie-kameru umístěnou na přední straně zařízení. V nejjednodušším případě si prostě vyžádáme libovolný zdroj obrazu:

```
try {
  let options = {video: true};
  let stream = await navigator.mediaDevices.getUserMedia(options);
} catch (e) {
  // není žádná kamera, případně uživatel nepovolil přístup
}
```

Funkce je asynchronní a je pravděpodobné, že na její dokončení si počkáme. Prohlížeč v tuto chvíli musí uživateli ukázat dialog, ve kterém dojde nejen k odsouhlasení přístupu webové stránky ke kameře, ale také k výběru konkrétní kamery, pokud jich je v zařízení dostupných více. Proměnná **stream** pak bude obsahovat objekt typu **MediaStream**, který použijeme jako zdroj dat pro **<video>**:

```
let video = document.querySelector("video");
video.srcObject = stream;
video.play();
```

Značku **<video>** v tomto případě vůbec nemusíme ukazovat – je pro nás jen nezbytný spojovací článek mezi streamem z kamery a canvasem, ve kterém výsledná data zobrazujeme a upravujeme.

Profíci: kdy s canvasem narazíme a jak z toho ven

V této podkapitole si ukážeme některé zapeklité situace, na které můžeme při práci s canvasem narazit. Jejich řešení je často snadné, ale ne vždy na první pohled zřejmé.

Tainting

V páté kapitole jsme se dozvěděli, že v JavaScriptu není obecně možné přistoupit k datům získaným z jiné domény, než je doména aktuální stránky. Jedná se o mechanismus **Same Origin Policy** a jeho hlavním účelem je ochrana proti útokům CSRF, tj. situacím, kdy vstoupíme na stránku útočníka a náš prohlížeč jí poskytne citlivé údaje. Na ochranu SOP můžeme narazit trochu nezvykle i při práci s canvasem, aniž bychom přímo vykonávali HTTP požadavky prostřednictvím **XHR** nebo **fetch**.

Začněme obyčejným HTML obrázkem (značkou ****), kterému nastavíme zdrojovou adresu s jiným originem (tento důležitý termín je definován rovněž v páté kapitole), než je origin současné stránky. Prohlížeč jej stáhne a zobrazí, ale protože se jedná o potenciálně citlivá data, nesmí je (bez souhlasu vzdáleného serveru) dát k dispozici JavaScriptu. Můžeme se o to pokusit použitím již známé metody **drawImage** – a v takovou chvíli se obrázek v canvasu sice vykreslí, ale prohlížeč si zároveň canvas vnitřně označí jako **tainted** (něco jako *otrávený*). To znamená, že přestanou fungovat veškeré mechanismy, kterými bychom se mohli pomocí JavaScriptu dívat na obrazová data v canvasu obsažená. Zahrnuje to jak zmíněné metody **toDataURL** a **toBlob**, tak i možnost čtení jednotlivých pixelů pomocí metody **getImageData**. Dokud canvas, který je *tainted*, zcela nevyprázdníme, nebudeme jej moci plnohodnotně používat.

Za toto bezpečnostní omezení jsme rádi, ale v některých situacích je zbytečně striktní. Víme, že SOP lze obejít, pokud vzdálený server vydá explicitní souhlas s tím, aby jeho data byla přístupná JavaScriptu. U HTML obrázků musíme nejprve přidat **crossOrigin**, ať už pomocí atributu nebo vlastnosti:

```

```

```
let image = new Image();  
image.crossOrigin = "anonymous";
```

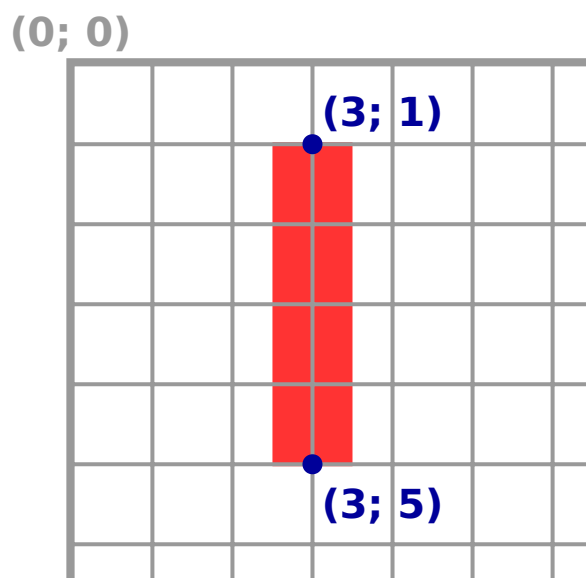
Povolené hodnoty jsou dvě: **anonymous** (do požadavku nejsou přidány cookies) a **use-credentials** (cookies jsou přítomny). Tím prohlížeč v požadavku na obrázek pošle také hlavičku **Origin**. Pokud následně server do HTTP odpovědi přidá potřebné CORS hlavičky (stejně jako na konci páté kapitoly), canvas s tímto obrázkem nebude *tainted* a můžeme s ním pracovat plnohodnotně.

Antialiasing

V podkapitole pro zelenáče jsme viděli, jak do canvasu kreslit různé geometrické útvary a křivky. V rámci tahu štětcem (posloupnost příkazů začínající metodou **beginPath**) musí prohlížeč danou křivku převést na jednotlivé pixely v canvasu – této operaci se říká *rasterizace*. Pixely jsou umístěny v pravidelné mřížce a během rasterizace se rozhoduje, který z nich bude mít jakou barvu. Tato činnost je docela komplexní a u složitějších křivek ji vlastně ani nelze vykonat zcela přesně. Můžeme si to představit na jednoduché diagonální čáře; při její rasterizaci na čtverečkováném papíře vidíme, že vzniklý útvar je *zubatý*.

V HTML canvasu prohlížeče automaticky během rasterizace provádí tzv. *antialiasing*. Při této operaci se snaží vyhnout zmiňovaným zubatým hranám tak, že pixely poblíž hranic vykreslované oblasti zobrazí barvou, která je někde mezi barvou výplně a barvou pozadí. Pokud do bílého canvasu kreslíme černý útvar, jeho hrany budou při bližším zkoumání šedé. Tato nepřesnost pak pro lidské oko zvýší iluzi toho, že hrana je hladká a přesná.

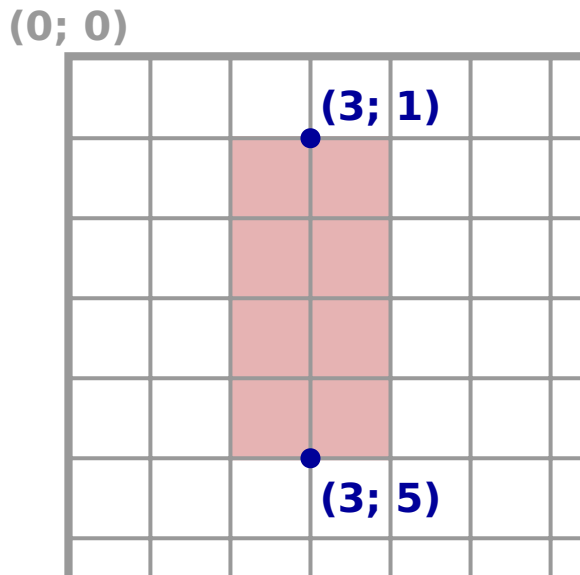
V naprosté většině případů je antialiasing užitečný a chtěný. Existuje však jeden scénář, kdy nám vadí – a přitom jej nelze vypnout. Jedná se o kresbu čáry, která je zcela svislá či vodorovná, má celočíselné souřadnice a lichou tloušťku. Jeden obrázek je v tomto případě lepší, než tisíc slov: pokusme se vykreslit krátkou svislou čáru, tlustou jeden pixel, spojující body (3; 1) a (3; 5).



Obrázek: Cílová tenká (jednopixelová) čára

Každý pixel si můžeme představit jako malý čtverec, který má v levém horním rohu své souřadnice. Pixel $(3; 1)$ je druhý shora a čtvrtý zleva. Jeho pravý dolní roh má souřadnice o jedničku vyšší – je to zároveň levý horní roh sousedního pixelu.

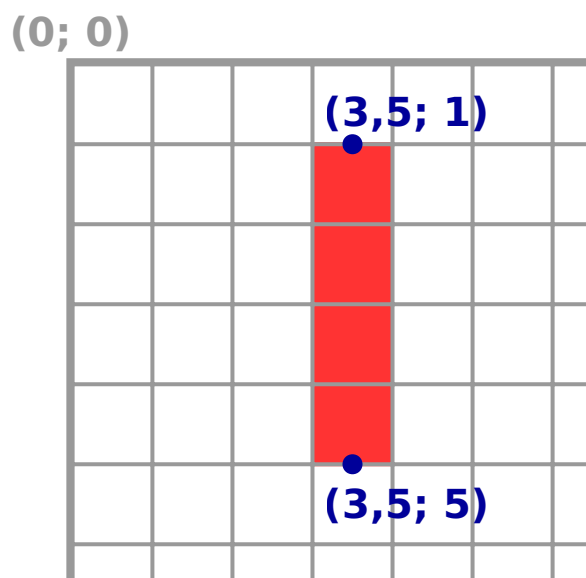
Podívejme se nyní na problémovou svislou čáru z bodu $(3; 1)$ dolů. Její tloušťka je jeden pixel, což znamená, že od svého matematického středu zabírá půl pixelu doleva a půl pixelu doprava. Do šířky by tedy měla pokrývat rozpětí od 2,5 do 3,5 pixelu. To ale nedává smysl, protože není možné zapnout na obrazovce (ani v canvasu) jen polovinu pixelu. Prohlížeč tento problém řeší antialiasingem, takže obarví oba dva pixely takto zasažené čarou, ale nastaví jim jen poloviční barvu. Namísto jednopixelové černé čáry pak vidíme čáru dvoupixelovou šedou.



Obrázek: Jednoplexová čára zabírá dva půlpixely

Víme, že antialiasing v canvasu nejde vypnout. Při pohledu na zvětšené pixely nás ale může napadnout snadný trik. Naším cílem je obarvit pixel (3; 1) a následně ty pod ním. Horizontálně to tedy znamená, že chceme obarvit rozpětí od třetího do čtvrtého pixelu. To odpovídá jednoplexové čáře, jejíž první souřadnice je 3,5! A canvas nám dovoluje používání neceločíselných souřadnic. Správně tedy ostrou svislou čáru (s lichou šířkou) vykreslíme tak, že ji posuneme o polovinu pixelu:

```
ctx.lineWidth = 1;
ctx.beginPath();
ctx.moveTo(3.5, 1);
ctx.lineTo(3.5, 5);
ctx.stroke();
```



Obrázek: Posun o polovinu pixelu doprava

Vysoká hustota pixelů

V posledních letech se často setkáváme se zobrazovacími zařízeními, které disponují vysokou hustotou pixelů (známe je např. pod obchodním názvem Retina). Tyto displeje mají extrémně malé pixely, takže se jich – v porovnání s běžnými displeji – na stejnou plochu vejde třeba čtyřnásobek. Díky tomu je možné dosáhnout perfektní ostrosti. Takto malé pixely ale neznamenaají, že pracujeme s násobným rozlišením. Můžeme si to představit třeba na písmu běžné výšky 16 pixelů, zobrazovaném na displeji s dvojnásobnou hustotou pixelů. Pokud bychom u takového displeje vykreslovali písmo vždy na 16 pixelů výšky, bude ve skutečnosti (v milimetrech) dvakrát menší, a tím pádem nečitelné.

Zařízení s vysokou hustotou proto používají trik: vykreslované objekty před rasterizací úměrně zvětší (typicky na dvojnásobek), takže písmo s nastavenou velikostí 16 pixelů jich zabere 32 a přitom je (v milimetrech) stejně velké, jako kdyby na běžném displeji zabralo pixelů 16.

Pro vývojáře to představuje jistou komplikaci, neboť nyní pracujeme se dvěma druhy pixelů:

1. *CSS pixely* neboli **logické pixely**, které odpovídají původnímu rozlišení. Když si koupíme displej Retina, počet logických pixelů zůstane stejný. Logické pixely používáme téměř všude: při specifikaci délkových jednotek v CSS, při nastavování rozměrů obrázků, v media queries.
2. *Hardwarové* neboli **fyzické pixely**, které odpovídají skutečným diodám tvořícím hardware displeje. S nimi pracuje až prohlížeč a operační systém, když obsah webové stránky rasterizují před vykreslením.

V JavaScriptu máme k dispozici informaci jen o logických pixelech, ale zároveň se ve vlastnosti `window.devicePixelRatio` dozvíme, kolikrát je počet fyzických pixelů větší než počet těch logických. Jinými slovy, displeje s vysokou hustotou pixelů mají `devicePixelRatio` větší než jedna.

Koncept logických a fyzických pixelů je navržen tak, abychom se o něj v naprosté většině případů nemuseli starat. Bohužel, práce s canvasem je jedno z těch míst, kdy nás hardwarové pixely zajímají. Představme si stránku s HTML obsahem šířky 640 pixelů (logických), ve které máme canvas (aby do stránky dobře pasoval, jeho šířka je též 640). Do canvasu jsme nakreslili obrázek a nyní jej chceme zobrazit na monitoru s `devicePixelRatio=2`. Znamená to, že prohlížeč musí canvas nejprve roztáhnout na šířku 1280 pixelů (hardwarových) a pak teprve vykreslit. Během roztahení dojde k nepěknému rozmazání způsobenému jednoduše tím, že si prohlížeč musí čtyřnásobný počet pixelů domyslet.

Tomuto nežádoucímu rozmazání můžeme předejít tím, že si canvas nachystáme v takové velikosti, která odpovídá počtu hardwarových pixelů. Zahrnuje to tři kroky:

1. Canvasu nastavíme rozměry (atributy `width` a `height`) dle fyzických pixelů.
2. Canvasu nastavíme velikost (CSS vlastnosti `width` a `height`) dle logických pixelů – chceme, aby dobře pasoval do stránky široké 640 logických pixelů.
3. Vše, co do canvasu kreslíme, musíme vytvářet dvakrát větší (neboť i počet pixelů v canvasu je větší).

Náš kód by měl ale stejně dobře fungovat jak pro zařízení s vysokou hustotou pixelů, tak pro ta běžná. Podívejme se, jak můžeme canvas pro takové vykreslování připravit zcela univerzálně, jen na základě znalosti **devicePixelRatio**:

```
// CSS pixely
let width = 640;
let height = 480;

let canvas = document.createElement("canvas");
canvas.style.width = `${width}px`;
canvas.style.height = `${height}px`;

// hardwarové pixely
canvas.width = width * devicePixelRatio;
canvas.height = height * devicePixelRatio;

canvas.scale(devicePixelRatio, devicePixelRatio);
```

Poslední řádek říká, že všechny následující vykreslovací operace mají být *zvětšeny* dle zadaného měřítka (resp. dle dvou – jednoho v ose X, jednoho v ose Y). Pokud pracujeme s běžným displejem, je **devicePixelRatio** rovno jedné, žádné zvětšování se neodehrává a CSS rozměry se shodují s fyzickými. U displejů s vysokým rozlišením vytvoříme canvas veliký, vykreslujeme jej do menšího prostoru, a tím se naše canvasové pixely přesně trečí do těch hardwarových.

Pozor – některé displeje mají hodnotu **devicePixelRatio** neceločíselnou. Musíme pak dát pozor na to, abychom po násobení zůstali u celých čísel, neboť canvas musí mít celočíselné rozměry.

JavaScript mimo prohlížeč

Většinu knihy jsme se věnovali JavaScriptu v prostředí webového prohlížeče, dokumentu HTML a rozhraní DOM. JavaScript je však univerzální jazyk, a tak není překvapivé, že ho lze používat i v jiných situacích. Často se použití bez prohlížeče týká serverového prostředí, ale ukážeme si, že se nám může hodit i pro běžné programátorské úkony zcela mimo provoz webových aplikací.

Proč bychom chtěli používat JavaScript tam, kde máme na výběr téměř všechny ostatní myslitelné programovací jazyky? Důvodů může být více: pokud už ovládáme JavaScript v prohlížeči, můžeme získané dovednosti znovupoužít na serveru. Může nám také vyhovovat koncept asynchronního programování, na kterém je JavaScript vystaven. A konečně, díky JavaScriptu připadá v úvahu možnost sdílení některého kódu mezi serverovou a klientskou částí aplikace. Tuto vlastnost se pokusíme využít i v úloze, kterou za malou chvíli vyřešíme.

Co to technologicky znamená, JavaScript bez prohlížeče? Potřebujeme implementaci jazyka samotného a k tomu takovou sadu rozhraní, abychom dokázali přistupovat k souborům, obrazovce, síti a dalším komponentám počítače. Zdaleka nejpopulárnější možností je v tomto směru projekt Node.js, který staví na jádře V8 (stejně jako prohlížeče Chrome a Edge) a přidává k němu bohatou sadu vestavěných knihoven. Výsledek je k dispozici pro všechny operační systémy a navíc zdarma (open source). Můžeme si jej stáhnout, nainstalovat a pak spouštět pomocí příkazu **node** v příkazové řádce.

Úloha

Vytvořte základ HTTP serveru, který bude umožňovat registraci uživatelských účtů. Založené účty evidujte v souboru **accounts.json**.

Řešení

```
// kapitola-12.js
import * as fs from "node:fs";
import * as http from "node:http";
import validateUsername from "../validate-username.js";

let accounts = [];
try {
  let data = fs.readFileSync("accounts.json");
  accounts = JSON.parse(data);
} catch (e) {}

function processRequest(req, res) {
  if (req.method !== "POST") {
    res.writeHead(204);
    res.end();
    return;
  }

  let body = "";
  req.on("data", chunk => body += chunk.toString());

  req.on("end", () => {
    let data = JSON.parse(body);

    if (validateUsername(data.username)) {
      accounts.push(data);
      res.writeHead(200);
      let str = JSON.stringify(accounts);
      fs.writeFile("accounts.json", str, e => {});
    } else {
      res.writeHead(400);
    }

    res.end();
  });
}
```

```

}

const server = http.createServer(processRequest);
server.listen(8000);

```

```

// validate-username.js
export default function(username) {
  return username.match(/^[a-z]/);
}

```

```

// package.json
{
  "type": "module"
}

```

Přestože se jedná o JavaScript, můžeme být překvapeni některými odlišnostmi od kódu, který jsme si v minulých kapitolách navykli psát pro prohlížeč. Začneme malinkatým souborem **package.json**, který je v ukázce až úplně na konci a který bývá součástí každého projektu pro Node.js. Je určen pro různá metadata naší aplikace, výčet závislostí a další konfiguraci. My do něj do začátku přidáme jedinou konfigurační položku ("**type**": "**module**"), která říká, že kód plánujeme dělit do souborů pomocí ES modulů (poprvé jsme je potkali v osmé kapitole). Prostředí Node.js je relativně staré a historicky nabízí i odlišný způsob členění kódu do knihoven (říká se mu *CommonJS require*). Protože je ale jedním z našich cílů možnost sdílení kódu mezi serverem a prohlížečem, rozhodně chceme používat ES moduly, tj. klíčová slova **export** a **import**.

Samotný kód serveru je v souboru **kapitola-12.js**. Rozdělme si jej do menších částí. Hned na začátku importujeme potřebné knihovny:

```

import * as fs from "node:fs";
import * as http from "node:http";
import validateUsername from "../validate-username.js";

```

První dva importy odpovídají vestavěným knihovnám, které jsou součástí Node.js. Proto jejich identifikace začíná tzv. pseudo-protokolem **node:** – a hned zde vidíme první rozdíl oproti klientskému JavaScriptu, ve kterém lze importovat jen soubory z webových adres (HTTP/HTTPS), případně relativní vůči aktuálnímu skriptu (s tečkou na začátku). Třetí import je naše vlastní miniaturní

knihovna, která obsahuje jedinou funkci **validateUsername**. Tu plánujeme použít pro kontrolu zadaného uživatelského jména. Do vlastního souboru jsme ji umístili právě proto, abychom tento mohli následně importovat i do HTML dokumentu a kontrolní funkci mohli používat i při klientské kontrole před odesláním formuláře (viz kapitolu 4).

Namísto opravdové databáze budeme uživatelské účty spravovat v obyčejném souboru. Kód našeho serveru proto začíná načtením existujících dat z tohoto souboru. K tomu se hodí funkce **readFileSync**. Blok try-catch je přítomen pro případ, že by soubor neexistoval:

```
let accounts = [];  
try {  
  let data = fs.readFileSync("accounts.json");  
  accounts = JSON.parse(data);  
} catch (e) {}
```

Dostáváme se k hlavní části aplikace. HTTP server je dlouho-běžící program (spustíme jej a očekáváme, že bude spuštěn stále, dokud jej sami ručně nevypneme), jehož úkolem je zpracovávat požadavky přicházející po síti. V rámci standardní knihovny Node.js jsme získali modul **http**, který většinu této práce udělá za nás. Stačí pak dodat vlastní funkci, která jako parametr získá data příchozího požadavku a jejímž úkolem bude vyrobit odpověď. Ve vzorovém řešení je pojmenovaná **processRequest** a server ji zavolá pro každý příchozí HTTP požadavek. O jejích parametrech se můžeme dočíst v oficiální dokumentaci Node.js.

Nejprve musíme rozpoznat, jestli se skutečně jedná o požadavek na registraci uživatelského účtu. Naše kontrola je jednoduchá:

```
if (req.method !== "POST") {  
  res.writeHead(204);  
  res.end();  
  return;  
}
```

Pro jednoduchost jen ověřujeme, zdali je požadavek realizován HTTP metodou **POST**. V praxi bychom typicky chtěli ještě kontrolovat URL, formát dat (hlavičku **Content-Type**) a další. Nevhodné požadavky zodpovíme stavovým kódem 204

(tzv. *No Content*; tento předdefinovaný stav informuje klienta, že server na požadavek nemá žádnou odpověď) a metodou `res.end()` odešleme zpět klientovi.

Následuje několik řádků věnovaných zpracování těla požadavku:

```
let body = "";
req.on("data", chunk => body += chunk.toString());
req.on("end", () => { ... })
```

Předpokládáme, že klient nám data posílá ve formátu JSON. V kontextu HTTP serveru je ovšem tělo požadavku představováno prostým řetězcem, který navíc může po síti přicházet v menších kouscích (paketech). Proto tyto části postupně přidáváme do velkého řetězce. Teprve když data od klienta dorazí všechna, lze pokračovat dále.

Můžeme si přitom všimnout, že funkce `processRequest` je asynchronní. Její vykonávání už dávno skončilo, server může zpracovávat další požadavky, ale my mezitím čekáme na data. V knihovně `node:http` se o datech dozvídáme prostřednictvím událostí – obdobně jako v rozhraní DOM metodou `addEventListener`. Ve světě serverového JavaScriptu ale není žádný strom dokumentu, a proto se s událostmi pracuje trochu odlišně; mimo jiné i jinak pojmenovanými metodami. V tomto případě přidáváme posluchač metodou `on` a zajímají nás dvě události: příchod další části dat (událost `data`) a ukončení HTTP požadavku (událost `end`).

Poslední část funkce `processRequest` následně zpracuje získaná data:

```
let data = JSON.parse(body);

if (validateUsername(data.username)) {
  accounts.push(data);
  res.writeHead(200);
  let str = JSON.stringify(accounts);
  fs.writeFile("accounts.json", str, e => {});
} else {
  res.writeHead(400);
}

res.end();
```

Pro kontrolu získaných dat použijeme importovanou funkci **validateUsername**, která vrací pravdivostní hodnotu. Při neúspěchu jen vrátíme stavový kód 400 (tzn. *Bad Request*, tedy chyba klienta). Pokud kontrola dopadne dobře, přidáme získaná data do pole **accounts** a jeho obsah následně zapíšeme do souboru. Zápis tentokrát realizujeme asynchronní funkcí **writeFile**, která neblokuje další vykonávání kódu – to proto, abychom zápisem na disk neoddalovali odpověď klientovi a zpracování dalších požadavků. Třetí parametr pro **writeFile** je *callback*; funkce, která bude vykonána po zapsání a jako parametr dostane informaci o případné chybě.

Výše uvedený kód je funkční, ale rozhodně není robustní. Za domácí úkol je možné naimplementovat celou řadu jeho vylepšení:

1. Neprovádíme žádnou kontrolu duplicity uživatelských jmen.
2. Do pole **accounts** ukládáme veškerá příchozí data, aniž bychom se podívali, kolik jich je (a co je jejich obsahem).
3. Pokud v rámci registrace ukládáme také heslo, jistě bychom jej měli vhodným způsobem zabezpečit (uložit jej hashované a osolené).
4. Pokud selže kontrola uživ. jména, měli bychom v odpovědi klientovi sdělit také informaci o tom, co přesně bylo v poslaném jménu špatně.

Úplně na konci vzorového řešení zbývá samotné spuštění serveru:

```
const server = http.createServer(processRequest);
server.listen(8000);
```

Parametr pro funkci **listen** je číslo TCP portu, na kterém bude server poslouchat. Na závěr v rychlosti prohlédneme knihovnu **validate-username.js**, která se stará o kontrolu uživatelského jména:

```
// validate-username.js
export default function(username) {
  if (username.match(/^[a-z]/)) {
    return true;
  } else {
    return false;
  }
}
```

Jedná se o prostou (a dost možná ne zcela dostačující) kontrolu pomocí regulárního výrazu; s takovými jsme se již setkali ve čtvrté kapitole. Naši aplikaci můžeme nyní spustit např. příkazem

```
node kapitola-12.js
```

Protože v aplikaci nevypisujeme na obrazovku serveru žádné informace, budeme muset její spuštění a běh sledovat jinak. Připomeňme, že naše aplikace je navržena specificky pro příjem HTTP požadavků POST, které mají v těle JSON s klíčem **username**. Pro ověření tedy můžeme použít například nástroj **curl**:

```
curl -v --data '{"username":"test"}' http://localhost:8000
```

Pokud je vše spuštěno správně, dostaneme odpověď s kódem 200 (OK) a v souboru **accounts.json** se objeví nový záznam.

Co jsme se naučili

Po vyřešení jedenácté úlohy by měl čtenář chápat a ovládat:

- spuštění JS aplikace pomocí Node.js
- import funkcionality ze standardních knihoven Node.js
- koncepty obsluhy HTTP požadavků

Zelenáči: npm

Při používání Node.js se nejčastěji potkáme s programem **node**, který slouží ke spuštění JavaScriptového kódu. Další důležitou součástí ekosystému Node.js je nástroj **npm** (*Node Package Manager*), který slouží k instalaci a správě knihoven – typicky těch, na kterých náš projekt závisí.

Viděli jsme, že pomocí klíčového slova **import** můžeme připojit kód z dalších souborů našeho projektu, stejně jako z vestavěných knihoven. Zcela jsme ale zatím přeskočili miliony a miliony řádků kódu, které vytvořili ostatní programátoři a dali k dispozici k všeobecnému používání online. Pojdme zkusit do našeho serveru přidat barevný výpis toho, co server zrovna provádí.

Náš HTTP server je spuštěn jako běžný program v rámci příkazové řádky, takže z něj můžeme snadno psát do terminálu. Funkce `console.log`, kterou známe především jako nástroj pro přístup k vývojářským nástrojům v prohlížeči, je k dispozici i v Node.js – její výstup si můžeme přecíst na příkazové řádce, kde je program spuštěný. Zbývá jen dodat požadované barvy.

Při výpisu textu do terminálu se barvy definují pomocí speciálních znaků vložených do vypisovaného řetězce. Mohli bychom je nastudovat a text s nimi obohatit, ale jistě bude snazší použít hotovou knihovnu. Pro naše potřeby se ideálně hodí **chalk**, která je nejen veřejně dostupná na GitHubu, ale zároveň je připravená pro Node.js na webu <https://www.npmjs.com/>. Odtud ji můžeme snadno získat právě pomocí programu **npm**.

V příkazové řádce, tam kde máme připravený HTTP server, napíšeme:

```
npm install chalk
```

Program **npm** stáhne zadanou knihovnu a uloží ji do podadresáře **node_modules**. Toto umístění je předmětem domluvy mezi programy **npm** a **node**, neboť právě v **node_modules** hledá Node.js knihovny, které nejsou ani vestavěné, ani netvoří naši aplikaci. Znamená to, že následně v souboru **kapitola-12.js** můžeme v záhlaví přidat další import:

```
import chalk from "chalk";
```

Všimněme si, že se nejedná ani o vestavěnou knihovnu (název nezačíná **node**), ani o součást aplikace (název nezačíná tečkou). Node.js proto tento modul hledá v **node_modules**, kam jsme jej před chvílí nainstalovali.

S knihovnou **chalk** můžeme *obarvit* řetězce, které plánujeme vypisovat. Začneme posledními řádky serveru:

```
const server = http.createServer(processRequest);
const port = 8000;
server.listen(port);
console.log(chalk.green(`Server listening on port ${port}`));
```

Pro sledování aktivity serveru ještě přidáme výpis tam, kde úspěšně zpracováváme požadavky:

```

if (validateUsername(data.username)) {
  accounts.push(data);
  res.writeHead(200);
  let str = JSON.stringify(accounts);
  fs.writeFile("accounts.json", str, e => {});
  console.log(chalk.green(`Account ${data.username} created`));
} else {
  res.writeHead(400);
  console.log(chalk.red(`Invalid username ${data.username}`));
}

```

Barevný výpis pomůže čitelnosti a stál nás jen minimum úsilí. Spokojeni ale ještě nejsme. Pokud bychom nyní chtěli náš projekt spustit na jiném počítači, bude pro jeho spuštění chybět před chvilkou nainstalovaná knihovna **chalk** v adresáři **node_modules**. Je proto nutné dát nějakým způsobem najevo, že ke spuštění tohoto HTTP serveru je potřeba nejprve zmíněnou závislost doinstalovat.

Tím se znovu dostáváme k malému konfiguračnímu souboru **package.json**. Právě v něm je obvyklé definovat, které všechny knihovny jsou pro běh aplikace nezbytné. Stačí jen, abychom přidali při instalaci parametr:

```
npm install -S chalk
```

Program **npm** teď nejen závislost stáhne a uloží na správné místo, ale zároveň její jméno zaznamená v **package.json**. Až pak náš kód budeme chtít spustit na jiném počítači, napíšeme prostě **npm install** (případně zkráceně **npm i**) a program **npm** namísto jedné knihovny stáhne a nainstaluje všechny, které budou v souboru **package.json** uvedeny.

Je dobré zmínit, že ačkoliv je **npm** určen především pro správu a instalaci závislostí, nabízí značné množství další funkcionality. Umí nás informovat o nových verzích používaných modulů, můžeme s ním instalovat knihovny jako spustitelné programy a v neposlední řadě nám dovoluje nahrávat naše vlastní knihovny na web npmjs.com.

Koumáci: tooling

Pojďme si nyní ukázat další situace, ve kterých se může JavaScript mimo prohlížeč hodit. Začneme úlohou, která je zkušenějším programátorům dobře známá: *testování*. Využijeme toho, že Node.js obsahuje vestavěnou podporu pro spouštění unit testů.

Testy

Automatizované testování je užitečná a zajímavá kapitola vývoje software. Na toto téma se píše obsáhlé knihy; my se omezíme na konstatování, že námi produkováný kód zpravidla kontrolujeme tak, že jej sami zkoušíme používat a pozorujeme, zdali se chová dle očekávání. Myšlenka automatizovaného testování říká, že pokud dokážeme formálně popsat, co od kódu očekáváme, můžeme pak následně nechat počítač, aby funkčnost kódu ověřil za nás.

Nejsnazší formou testování jsou tzv. **jednotkové testy** (anglicky *unit testing*). Při něm jednoduše nachystáme některé funkce z naší aplikace, opakovaně je vykonáváme s různými parametry a kontrolujeme, zdali jsou návratové hodnoty korektní. Za tímto účelem se nejlépe hodí funkce, které veškerý svůj vstup přijímají formou parametrů a výsledek své práce vrací výhradně pomocí návratové hodnoty. V úloze této kapitoly se pro potřeby testování docela dobře hodí sdílená funkce `validateUsername`.

Spouštění testů v prostředí Node.js je velmi snadné. Nejprve připravíme jeden či více *testových souborů*, které nejsou nezbytnou součástí aplikace, ale ve kterých některé aplikační komponenty otestujeme. Tyto testové soubory musí mít název, který začíná či končí (před příponou) slovem *test*. Pojďme založit nový soubor **kapitola-12.test.js**:

```
// kapitola-12.test.js
import test from "node:test";
import assert from "node:assert";
import validateUsername from "../validate-username.js";
```

K vytvoření testu potřebujeme tři základní stavební kameny:

1. Funkci **test**, která slouží pro definici testu.

2. Objekt **assert**, který slouží pro porovnání očekávané a skutečné hodnoty námi testované funkce.
3. Aplikační funkci, kterou plánujeme testovat.

Samotná definice testu funguje tak, že zavoláme funkci **test**, předáme jí název testu a malou (typicky anonymní) funkci, která testování provede. V ní budeme volat **validateUsername** a pomocí **assert.strictEqual** ověřovat, že se návratová hodnota shoduje s tou, kterou očekáváme:

```
test("funkce validateUsername", () => {
  assert.strictEqual(validateUsername("test"),    true);
  assert.strictEqual(validateUsername("0test"),   false);
  assert.strictEqual(validateUsername("123 ahoj"), false);
  assert.strictEqual(validateUsername(""),         false);
  assert.strictEqual(validateUsername(),           false);
  assert.strictEqual(validateUsername("a b"),      true);
  assert.strictEqual(validateUsername("a"),        true);
});
```

Dobré jednotkové testy pokrývají spoustu přípustných i nepřípustných variant volání funkce. Často bývá dokonce zvykem vytvořit tyto testy dříve, než samotnou implementaci – protože o tom, jak se má funkce chovat, míváme představu před tím, než ji napíšeme. Takovému postupu se říká TDD: **Test-Driven Development**.

Jakmile máme test připravený, můžeme jej spustit. V příkazové řádce napíšeme:

```
node --test
```

Pokud jsme postupovali přesně dle kódu v knize, uvidíme výpis spouštěných testů a informaci o tom, zdali všechny proběhly úspěšně, tj. zdali všechna volání funkce **assert.strictEqual** dostala identické parametry. Jenže ouha! Náš test skončil chybou a podle výpisu je vidět, na kterém řádku se tak stalo. Na základě toho snadno poznáme, které volání **validateUsername** nedopadlo dle očekávání.

Jedná se o variantu **validateUsername()**, kdy funkci nepředáme žádný parametr, tj. její lokální proměnná **username** bude mít hodnotu **undefined**. Těžko na ní tedy zavoláme metodu **match**. Přitom se zdaleka nejedná o umělý případ; stalo by se

tak pokaždé, kdyby našemu serveru přišel požadavek, v jehož JSON datech nebude přítomen klíč **username**. Díky testu jsme tedy odhalili skutečnou chybu v našem kódu!

Poučení tímto nezdarem upravíme implementaci funkce **validateUsername**:

```
export default function(username) {  
  if ((username || "").match(/^[a-z]/)) {  
    return true;  
  } else {  
    return false;  
  }  
}
```

Přidali jsme operátor *nebo* a tím zařídili, že pokud je parametr *falsy value*, použijeme místo něj prázdný řetězec (který má metodu **match**). A skutečně, testy nyní prochází bez chyb:

```
$ node --test  
✓ funkce validateUsername (0.950831ms)  
i tests 1  
i suites 0  
i pass 1  
i fail 0  
i cancelled 0  
i skipped 0  
i todo 0  
i duration_ms 44.574717
```

Bundling

Díky mohutnému ekosystému knihoven v rámci Node.js a npm je s pomocí JavaScriptu naprogramováno mnoho nástrojů, které nám při vývoji mohou různým způsobem pomáhat.

V osmé kapitole jsme ukázali, že objemnější kód je užitečné rozdělit do více souborů (a propojit direktivami **import** a **export**). Pro programátora je členění klíčové pro čitelnost a organizaci aplikace. Naši uživatelé však o takovou skladbu

nestojí, neboť jejich prohlížeč musí každý jednotlivý soubor naší aplikace získat individuálním HTTP požadavkem. Je proto běžné, že hotovou (modularizovanou) aplikaci pro potřeby provozu na webu *zabalíme* do jediného souboru, který připojíme k HTML dokumentu.

Nejedná se o balení ve smyslu komprimace. Výsledkem musí být stále JavaScript, ale upravený tak, aby celá funkcionality byla dostupná v jediném souboru. Něco takového nebudeme dělat ručně, ale použijeme k tomu existující balicí program, tzv. **bundler**.

Existuje jich celá řada, my si tento scénář předvedeme pomocí nástroje **esbuild**. Není náhodou, že je taktéž dostupný prostřednictvím nástroje **npm**. Jeho instalace je tedy jednoduchá:

```
npm install -g esbuild
```

Tentokrát jsme použili parametr **-g** (global), který upravuje chování **npm**:

1. Knihovna se neinstaluje do **node_modules**, ale do systémového adresáře, odkud je dostupná pro všechny uživatele.
2. Informace o knihovně se nezapiše do **package.json**. To je v pořádku, neboť **esbuild** nepotřebujeme k běhu našeho serveru. Vlastně nás žádný server v tuto chvíli ani nezajímá. Program **esbuild** chceme používat pro potřeby klientského JavaScriptu nezávisle na tom, má-li naše aplikace nějakou komponentu v Node.js.

V této podkapitole nemáme žádný kód, který bychom chtěli nebo mohli *bundlovat*. Program **esbuild** nicméně můžeme vyzkoušet třeba na kódu z osmé kapitoly, který jsme poprvé rozdělili do více souborů:

```
esbuild --bundle kapitola-8.js --outfile=kapitola-8.bundle.js
```

Vzniklý balíček, pojmenovaný **kapitola-8.bundle.js**, pak můžeme normálně připojit k HTML dokumentu. Jeho obsah je sice JavaScript, ale my nemáme potřebu do něj zasahovat. To proto, že při následných úpravách *zdrojových souborů* (**kapitola-8.js**, **comment.js**) budeme chtít pomocí **esbuildu** opět balíček přegenerovat. Tím bychom o případné ruční zásahy do něj přišli.

Profíci: alternativy k Node.js

Přestože Node.js nabízí pohodlný prostor pro JavaScriptové programování mimo prohlížeč, není jediným nástrojem svého druhu. Existuje několik alternativních možností, které se liší použitými technologiemi, výkonem, množstvím implementované funkcionality a obecně řečeno přístupem k DX (anglicky *developer experience*). Jinými slovy, všechny tyto nástroje soupeří o přízeň nás, programátorů. V této kapitole si některé z nich představíme. Nemáme sice prostor pro detailní ukázky, ale povíme si alespoň o jejich základních konceptech a rozdílech oproti Node.js.

Deno

Prvním a hlavním konkurentem pro Node.js je projekt *Deno*. Je poměrně nezvyklé, že jeho autorem je Ryan Dahl, tedy původní autor Node.js. V roce 2018 dospěl k závěru, že jím odstartovaný projekt Node.js z roku 2009 nesplňuje moderní požadavky na programovací nástroj, a rozhodl se vytvořit alternativu.

Deno stojí na stejné implementaci JavaScriptu (V8), ale jeho další funkcionality je implementována v jazyce Rust. Zprovoznění a vývoj projektu v Deno je v několika aspektech jednodušší než v Node.js:

- Program Deno není nutné instalovat, jedná se o jediný spustitelný soubor.
- Standardní knihovna (funkce pro práci se soubory, sítí, ...) není součástí staženého programu; namísto toho je při běhu programu stahována z internetu, a tak je vždy aktuální.
- Součástí Deno není koncept **node_modules** ani program **npm**. Nedochází k žádné instalaci závislostí, veškerý kód je importován – stejně jako v prohlížeči – z adres HTTP/HTTPS.
- Pokud bychom náš zdrojový kód psali v TypeScriptu (více o tom v příští kapitole), můžeme jej v Deno pustit rovnou, bez transpilace do JavaScriptu.

V prostředí Deno je značná snaha o znovupoužívání těch API, která známe z klientského JavaScriptu (tj. z webového prohlížeče). V případě úlohy s HTTP serverem bychom tak v rámci funkce **processRequest** pracovali s existujícími objekty typu **Request** a **Response**, které jsou součástí standardu **fetch**.

Deno láká vývojáře také doplňkovými službami. Jedná se zejména o *Deno Deploy*, službu hostingu pro servery a aplikace napsané v Deno. Je to zajímavá a snadná alternativa k provozu vlastních linuxových serverů. Další nabídkou ze světa Deno je distribuovaná databáze KV, navržená jako obecné datové úložiště právě pro aplikace provozované v Deno Deploy.

Bun

Nejnovějším zápasníkem v ringu JavaScriptových prostředí je Bun, který spatřil světlo světa v roce 2023. Jeho definující vlastností je výkon: v rychlosti většiny testovaných scénářů předčí nejen relativně starý Node.js, ale i novější Deno. Bun používá JavaScriptovou implementaci JavaScriptCore od firmy Apple, kterou jinak známe z prohlížeče Safari; jeho další části jsou napsány v jazyce Zig.

Bun, stejně jako Deno, dovede přímo vykonávat soubory psané v TypeScriptu. Taktéž si zakládá na podpoře standardních prohlížečových API (např. **WebSocket**, **fetch**, ...) tam, kde Node.js potřebuje vlastní řešení, případně knihovnu staženou z **npm**.

S ohledem na nízký věk projektu Bun zatím není zřejmé, které budou definující odlišnosti od ostatních prostředí. Při pohledu na jeho výkonové parametry je nicméně zajímavým experimentem, který rozhodně stojí za sledování.

Když JavaScript nestačí

V předposlední kapitole už nebudeme řešit konkrétní úlohu, ale ukážeme si další směry, kam se v rámci prozkoumávání klientských webových technologií můžeme vydávat. Jakkoliv je totiž JavaScript užitečný, samotný nám nedokáže poskytnout vše, co bychom mohli potřebovat. Podívejme se tedy na několik populárních knihoven a nadstaveb, se kterými se v praxi můžeme setkat.

Zelenáči: React

Pro mnohé frontendové vývojáře je knihovna React stejně důležitá, jako JavaScript samotný. Jedná se o nástroj původně vyvinutý pro interní potřeby projektu Facebook, který se následně rychle rozšířil po celém internetu. React je určen k tvorbě stromu stránky ze vstupních dat. Takovou úlohu jsme v této knize řešili několikrát – použili jsme k tomu buď vestavěný HTML parser (tj. vlastnost `innerHTML`), nebo individuální metody dostupné v rozhraní DOM (jako např. `document.createElement`). React volí přístup koncepčně někde mezi těmito dvěma variantami: strom stránky vytváříme deklarativně (jako kdybychom rovnou psali HTML), ale atributy a textové uzly můžeme generovat rovnou z JavaScriptových dat (jako kdybychom nastavovali atributy DOM objektům).

React staví zejména na těchto konceptech:

- HTML výstup definujeme pomocí **značkovacího jazyka JSX**, který HTML připomíná. Jedná se o mix HTML a JavaScriptu; podobá se také trochu *template literals* (tyto řetězce jsme viděli v páté kapitole). React lze používat i bez JSX, ale připravujeme se tím o jisté množství funkcionality. Při naší ukázce vyzkoušíme React včetně JSX.
- Jednotlivé části kódu organizujeme do tzv. **komponent**, které zodpovídají za jednotlivé části stromu dokumentu. Úkolem komponent (a úkolem Reactu jako takového) je prezentace dat. Tato jsou komponentám v jazyce JSX předávána pomocí atributů, které mohou – v porovnání s HTML – obsahovat

libovolné datové typy. V jazyce JSX pak komponenty vypadají jako HTML značky a nápadně připomínají *Custom Elements* (devátá kapitola). Komponenty jsou typicky implementovány jako JavaScriptové funkce.

- Při použití Reactu necháváme knihovnu vygenerovat výstup při libovolné změně ve vstupních datech, což zní jako zbytečná práce. React ale vzniklé HTML prvky negeneruje rovnou ve stránce, ale v rámci tzv. **virtual DOM** – svého virtuálního dokumentu. Do stránky pak promítne jen ty úpravy, které znamenají změnu v dokumentu opravdovém.

React si můžeme vyzkoušet na úloze z šesté kapitoly, ve které implementujeme zobrazování výsledků hledání. Vystačíme si s minimálním HTML dokumentem:

```
<!-- index.html -->
<body>
  <main></main>
  <script src="index.js"></script>
</body>
```

Celé HTML plánujeme vytvářet pomocí Reactu. Budeme k tomu používat jazyk JSX, takže výsledný skript **index.js** si na závěr necháme vygenerovat. Místo toho začneme vstupním bodem aplikace v souboru **index.jsx**:

```
// index.jsx
import { createRoot } from "react-dom/client";
import App from "./App.jsx";

let main = document.querySelector("main");
createRoot(main).render(<App />)
```

Většina vstupních bodů reactových aplikací vypadá obdobně: pomocí funkce **createRoot** označíme místo ve stránce, do kterého necháme React generovat obsah prostřednictvím komponent. Zpravidla nepoužíváme celé **<body>**, ale jen nějakou pod-značku (jako zde **<main>**). To proto, aby nám zůstala možnost některé další části dokumentu spravovat bez použití Reactu.

Na druhém řádku pak importujeme hlavní komponentu, která zastřešuje celou aplikaci. V parametru funkce **render** již vidíme jazyk JSX v akci, když tuto komponentu předáváme jako parametr. Podívejme se na její implementaci:

```
// App.jsx
import { useState } from "react";
import Results from "./Results.jsx";

export default function App() {
  let [results, setResults] = useState({});

  async function onSubmit(e) {
    e.preventDefault();
    let query = encodeURIComponent(e.target.elements.query.value);
    let response = await fetch(`/search?query=${query}`);
    let results = await response.json();
    setResults({results, query});
  }

  return <>
    <h1>Hledání</h1>
    <form onSubmit={onSubmit}>
      <label>
        Hledaný výraz: <input type="text" name="query" />
      </label>
      <label>
        <button></button>
      </label>
    </form>
    <Results data={results} />
  </>;
}
```

Nenechme se zastrašit větším množstvím nezvyklých konstrukcí a pojďme si je po částech rozebrat. Poslední část funkce **App** definuje výsledné HTML, resp. výsledné JSX. To odpovídá přístupu **komponenty jsou funkce, které transformují**

vstup na HTML. Každá komponenta musí vrátit jen jeden prvek, takže pokud chceme vykreslit více HTML (nadpis, formulář, výsledky), musíme je zabalit do společného *bezejmenného* obalu, který se v JSX zapisuje jako `<>...</>`.

Na konci HTML používáme další vlastní komponentu (`<Results>`), ke které se dostaneme za chvíli. Syntaxe se složenými závorkami odpovídá interpolaci hodnot z proměnných; zápis

```
<Results data={results} />
```

tedy znamená, že zde chceme vykreslit zadanou komponentu a předat jí obsah lokální proměnné **results**.

Samotné hledání provede funkce **onSubmit**, kterou pro jednoduchost implementujeme pomocí **fetch**. Je to posluchač události, který v JSX přidáváme atributem **onSubmit**. Uvnitř této funkce potřebujeme přistoupit k hodnotě z vyhledávacího pole. Nemůžeme použít běžné dotazování pomocí **querySelector**, neboť celou tvorbu stromu DOMu za nás řeší React. Využijeme proto klíč **e.target**, který odpovídá odesílanému formuláři. Jeho vlastnost **elements** pak dovoluje přistoupit k jednotlivým formulářovým prvkům pomocí jejich jména (tedy HTML atributu **name**). Výsledný dotaz je zaslán na server a po obdržení odpovědi se zavolá funkce **setResults**, která aktualizuje stav aplikace.

V tuto chvíli je dobré poznamenat, že React nabízí i další způsoby, jak se v rámci posluchače dostat k hodnotě z formulářového pole. Představené řešení je sice funkční, ale v praxi můžeme narazit na odlišné varianty. Zmíňme pro inspiraci dvě z nich:

1. Formulářovému poli `<input>` můžeme přidat posluchač na změnu hodnoty (DOM událost **input**). V rámci posluchače je pak zadaný text dostupný jako **e.target.value** a my si jej můžeme například uložit do lokální proměnné.
2. K jednotlivým Reactem vytvořeným HTML prvkům se můžeme dostat pomocí tzv. *referencí* vytvářených reactovou funkcí **useRef**. Lze si je představit jako samolepky, kterými pomocí atributu **ref** označíme prvky v JSX a díky nim se pak dostaneme k jim odpovídajícím DOM objektům.

Poslední specialita v souboru **App.jsx** je funkce **useState**. Připomeňme, že hlavní úkol Reactu je převést naše data na HTML. Ideálně bychom rádi, aby komponenta tuto operaci provedla automaticky vždy, když její vstupní data změníme.

Pouhá změna hodnoty v proměnné (jako např. **results**) k tomu ale nestačí, protože React se o takové změně nedozví. Zde vstupuje do hry funkce **useState**, která vytvoří navzájem provázanou dvojici proměnných: jednu pro hodnotu a jednu pro jí odpovídající funkci určenou ke změně hodnoty. Jakmile funkci zavoláme, hodnota se změní a *React dostane pokyn k překreslení*.

Mimochodem: využíváme zde zkrácený zápis definice objektu, který jsme poprvé potkali v osmé kapitole, konkrétně v její podkapitole pro zelenáče. Do funkce **setResults** tedy předáváme **{results:results, query:query}**.

Pokračujme ke komponentě **<Results>** v souboru **Results.jsx**:

```
// Results.jsx
import Result from "../Result.jsx";

export default function Results(props) {
  const { query, results } = props.data;

  if (!query) { return; }

  if (!results.length) {
    return <p>Tomuto dotazu nevyhovují žádné písně 😞</p>;
  }

  let items = results.map(item => <Result data={item} />);
  return <>
    <h2>Nalezené písně pro dotaz: {query}</h2>
    <ol>{items}</ol>
  </>;
}
```

Zde žádná velká překvapení nejsou. JSX atributy, které komponentě předáváme, jsou v odpovídající JS funkci dostupné jako vlastnosti objektu předaného v prvním parametru. Bývá obvyklé jej pojmenovat **props** (z anglického *properties*). Pro následné rozdělení dat do lokálních proměnných používáme destructuring (slyšeli jsme o něm v jedenácté kapitole).

Komponenta **<Results>** zastřešuje tři různé stavy (nebylo hledáno / nejsou výsledky / jsou výsledky), čemuž odpovídají jednotlivé podmínky. V případě nalezených výsledků vidíme cyklus realizovaný funkcí **map**. To znamená, že ve výsledném JSX interpolujeme proměnnou **items**, která je pole komponent typu **<Result>**. Ty jsou definovány v posledním souboru **Result.jsx**:

```
// Result.jsx
export default function Result(props) {
  let html = {__html: props.data.text};
  return <li>
    <a href={props.data.url}>{props.data.title}</a>
    <br/>
    <span dangerouslySetInnerHTML={html} />
  </li>;
}
```

Vzpomeňme si na šestou kapitolu, ve které jsme narazili na koncept **innerHTML** a jeho potenciální důsledky pro bezpečnost aplikace. React se nás snaží od podobného postupu odradit, takže přímé vložení HTML kódu z proměnné je nezvykle složité:

- Nejprve musíme nachystat objekt s klíčem **__html**, jehož hodnota je náš řetězec obsahující (potenciálně rizikové) HTML.
- Poté tento objekt předáme atributu **dangerouslySetInnerHTML**, jehož výmluvné označení varuje, že jde o nebezpečnou operaci.

Co dál? Napsaný kód je docela složitý a prohlížeč mu nerozumí. Jednak nezná syntaxi JSX, jednak neví, jak importovat z "**react-dom/client**" (vzpomeňme na serverové importy z minulé kapitoly). Budeme muset použít nějaký nástroj, který JSX převede na použitelný JavaScript.

Z minulé kapitoly známe program **esbuild** (používali jsme jej k *bundlingu*). Teď se nám hodí, že **esbuild** dokáže mimo jiné zpracovávat soubory JSX a také používat závislosti z adresáře **node_modules**. Použijeme proto **npm** a nainstalujeme několik balíčků:

```
npm i -g esbuild      # zpracování JSX, bundling
npm i react react-dom # součásti reactu
```

Nástroj **esbuild** instalujeme globálně, zatímco React a React-DOM lokálně do **node_modules**. Teď zbývá jen vygenerovat celou aplikaci:

```
esbuild --bundle index.jsx --jsx=automatic --outfile=index.js
```

Na závěr poznamenejme, že kompilace z JSX do JS je dnes dostupná prostřednictvím celé řady programů. Kromě zmiňovaného **esbuild** se často používají alternativy jako **babel**, **swc**, **rollup** nebo **tsc**, kterému se budeme věnovat v následující podkapitole.

Koumáci: TypeScript

Pojďme si nyní prohlédnout další jazyk, který je dnes s JavaScriptem neodmyslitelně spojen. TypeScript vznikl jako nadmnožina JavaScriptu, ve které můžeme k proměnným a funkcím volitelně přidávat informace o jejich datovém typu. Říká se jim **typové anotace** a můžeme je znát i z dalších jazyků, jako např. Python či PHP. V JavaScriptu však typové anotace neexistují, takže kód psaný v TypeScriptu nemůžeme rovnou předhodit prohlížeči. Podobně jako u JSX je nutné nejprve kód převést na JavaScript. Tentokrát je ale převod velmi snadný, protože typicky zahrnuje pouze smazání typových anotací.

A proč to celé vlastně chceme dělat? TypeScript je odpovědí na dynamické typování JavaScriptu, které je pro mnoho vývojářů až příliš flexibilní. Kvůli němu nelze rychlým pohledem na proměnnou či funkci odhadnout, jakých hodnot může nabývat. Snadno se pak vystavujeme riziku, že budeme nějakou hodnotu mylně považovat za jiný datový typ a dopustíme se různých chyb (např. výjimek způsobujících přerušení běhu programu).

S typovými anotacemi se v TypeScriptu setkáváme dvakrát. Když převádíme kód z TypeScriptu do JavaScriptu, kompilátor anotace nejen odstraní, ale také s jejich pomocí zkontroluje, zdali neporušujeme jimi definovaná pravidla – jestli třeba do proměnné označené jako *string* nekladáme číslo. Zároveň ale anotace používáme již při samotném psaní kódu, pokud si správně nastavíme editor. Ten může na základě typových informací rovnou označovat bloky problematického kódu, případně napovídat typy a názvy parametrů funkcí a podobně.

Zdrojový kód v TypeScriptu píšeme do souborů s příponou **.ts**. Nejčastěji se typové anotace objevují za dvojtečkou, za názvem proměnné či parametru funkce. Můžeme si to vyzkoušet na příkladu triviální sčítací funkce:

```
// test.ts
function add(a: number, b: number): number {
    return a+b;
}
```

Typové anotace jsme přidali jak k oběma parametrům, tak k funkci jako takové – říkáme tím, že její návratová hodnota je číslo. Na chování kódu anotace nemají žádný vliv, takže pokud bychom funkci předali dva řetězce, dojde k jejich (pravděpodobně chybnému) spojení. Proto bychom rádi, aby nás nějaká komponenta zavčas (na základě anotací) upozornila, že funkci voláme špatně. Nejprve napíšeme problémový kód:

```
let a = "protřepat";
let b = "nemíchat";
let c = add(a, b);
```

Dobře nastavené vývojové prostředí by nás už nyní mělo varovat, že funkci voláme chybně. My nyní ale stojíme o kontrolu spojenou s převodem do prostého JavaScriptu. K tomu použijeme oficiální kompilátor **tsc** (TypeScript Compiler). Nainstalujeme jej pomocí npm:

```
npm i -g typescript
```

Pro spuštění kompilace stačí předat jméno souboru s TypeScriptovým kódem:

```
tsc test.ts

test.ts:6:13 - error TS2345: Argument of type 'string' is not assignable to parameter of type 'number'.
```

Dostali jsme vynadáno přesně dle očekávání. Zároveň s tím ale vznikl soubor **test.js**, který obsahuje náš chybný kód bez anotací. Kompilátor nám tedy nebrání v psaní pochybného kódu, ale pokud mu k tomu dáme příležitost, upozorní nás na problematická místa.

Mimochodem: v minulých kapitolách a podkapitolách jsme používali nástroj **esbuild**, který mj. provádí *bundling* a transpilaci JSX. Zvědavého čtenáře možná napadne, zdali by pomocí **esbuild** nešel kompilovat i zdrojový kód v TypeScriptu. Odpověď je *ano, ale* – esbuild sice dovede odstranit typové anotace, ale neprovede při tom typovou kontrolu. Dostáváme tak jen polovinu funkcionality, kterou nám nabízí **tsc**.

Jazyk TypeScript je od počátku zamýšlen pro dobrovolné, pozvolné vylepšování JavaScriptového kódu. Pokud máme hotový projekt a rádi bychom do něj TypeScript zavedli, můžeme tak činit po malých krůčcích bez obav, že něco pokazíme. Program **tsc** lze vykonat i nad kódem, ve kterém typové anotace vů-

bec nejsou, nebo jsme je doplnili jen občasně. Součástí TypeScriptu je totiž tzv. **typová inference**, což znamená schopnost kompilátoru domyslet si v řadě případů datové typy podle toho, jakým způsobem s proměnnými pracujeme. Pokud například do proměnné při definici rovnou přiřadíme hodnotu, TypeScript odvodí její datový typ a bude s ním pracovat při dalších manipulacích s touto proměnnou:

```
let a = "ahoj";
a = 42; // Warning: Type 'number' is not assignable to type 'string'.
```

Stejně jako v minulé podkapitole, i nyní můžeme zkusit upravit vzorový kód z šesté kapitoly. Potřebujeme doplnit typové anotace tak, aby byl jasný datový typ každé proměnné a aby kompilátor nehlásil žádnou chybu. Hlavní data, se kterými pracujeme, jsou výsledky hledání. Na základě dohody s backendovou stranou aplikace víme, že se bude jednat o slovník s konkrétními položkami. Takový datový typ můžeme v TypeScriptu vyjádřit klíčovým slovem **interface**:

```
interface Song {
  url: string;
  title: string;
  text: string;
}

function buildSong(song: Song) {
  let item = document.createElement("li");
  item.innerHTML = `
```

```

    <a href="${song.url}">${song.title}</a>
    <br/> ${song.text}
  `;
  return item;
}

```

Dále přidáme typové informace do funkce, která je zodpovědná za zpracování a zobrazení výsledků hledání:

```

function showResults(xhr: XMLHttpRequest, query: string) {
  let results = document.querySelector("#results");
  if (!results) { return; }

  let songs = xhr.response as Song[];
  if (songs.length == 0) {
    results.replaceChildren("Dotazu nevyhovují žádné písně 😞");
    return;
  }

  let heading = document.createElement("h2");
  heading.textContent = `Nalezené písně pro dotaz: ${query}`;

  let ol = document.createElement("ol");
  results.replaceChildren(heading, ol);

  for (let i=0; i<songs.length; i++) {
    let song = buildSong(songs[i]);
    ol.append(song);
  }
}

```

TypeScript je obeznámen s vestavěnými objekty rozhraní DOM, takže u prvního parametru můžeme rovnou psát **xhr: XMLHttpRequest**. Další důležitá novinka se objevuje hned na prvním řádku funkce **showResults**, kdy je nutné věnovat více pozornosti metodě **querySelector**. Může se totiž snadno stát, že tato metoda nevrátí použitelnou hodnotu – když ve stránce takový prvek není nebo třeba

když uděláme překlep v zadaném selektoru. Pokud chceme s proměnnou **results** nadále pracovat, musíme mít jistotu, že je neprázdná. Proto přidáme na další řádek podmínku a vykonávání případně ukončíme.

Hodnota **xhr.response** může být jakákoliv, neboť z našeho kódu neumíme ovlivnit data, která server vygeneruje. Nemáme proto žádnou jistotu, že proměnná **songs** je opravdu pole struktur typu **Song**. Zápisem **as Song[]** kompilátoru říkáme, ať pro potřeby kontroly typů předpokládá, že tomu tak opravdu je. Kdybychom chtěli, mohli bychom ještě dodat explicitní kód, který by příchozí data prošel a zkontroloval.

Zbývá vylepšení poslední části kódu, kdy reagujeme na odeslání hledacího formuláře. Jedná se o posluchač události **submit**:

```
function onSubmit(e: Event) {
    e.preventDefault();
    let xhr = new XMLHttpRequest();
    let input = form.querySelector<HTMLInputElement>("[name=query]");
    if (!input) { return; }
    let query = input.value;
    let url = `/search?query=${encodeURIComponent(query)}`;
    xhr.responseType = "json";
    xhr.open("GET", url);
    xhr.send();
    xhr.addEventListener("load", e => showResults(xhr, query));
}
```

Kromě kontroly návratové hodnoty **querySelector** zde narážíme na další komplikaci. Různé HTML prvky mají různé JavaScriptové vlastnosti, takže jednou nám funkce **querySelector** může vrátit třeba obrázek (ten má mj. vlastnost **src**), zatímco jindy vrátí prvek **<input>** (který má vlastnost **value**). TypeScriptový kompilátor je v tomto směru bezradný, takže mu napovíme pomocí tzv. **typového parametru**. Zápisem **querySelector<HTMLInputElement>** říkáme, že počítáme s tím, že návratová hodnota bude zadaného typu (a proto je v pořádku u ní následně pracovat s vlastností **value**).

Úplně na konec přidáme posluchač na formulář a při tom si ukážeme ještě jeden syntaktický prvek ze světa TypeScriptu:

```
let form = document.querySelector("form");  
form.addEventListener("submit", onSubmit);
```

Víme, že funkce **querySelector** vrací buď HTML prvek, nebo **null**. Vykřičníkem říkáme, ať TypeScript možnost s **null** neuvažuje. I zde bychom mohli prostě použít podmínku (náš kód by byl robustnější!), ale vytvořili bychom tak komplikaci pro již upravený posluchač **onSubmit**. V něm se totiž, díky uzávěře, s proměnnou **form** taktéž pracuje. Kdyby její hodnota nebyla jistá, museli bychom s ní opatrně pracovat i uvnitř posluchače. My jako programátoři víme, že pokud se posluchač vykoná, proměnná **form** existuje – TypeScript to ale za nás vymyslet nedokáže.

Profíci: WebAssembly

Programovacích jazyků je kolem nás velké množství, a přesto je možné v rámci webového prohlížeče psát aplikační kód jen v JavaScriptu. Říkáme si, proč nemáme k dispozici i další možnosti jako například Python, Go nebo třeba Ruby. Jakkoliv by bylo zavedení dalšího jazyka do prohlížeče praktické, jedná se o příliš komplikovaný úkol – znamenalo by to, že všechny prohlížeče musí přijít s identickou implementací zvoleného jazyka a zároveň pro něj poskytovat všechna rozhraní, se kterými jsme v JavaScriptu navyklí pracovat (DOM, práce se sítí, Canvas, Web Audio, ...).

Namísto integrace nějakého dalšího konkrétního programovacího jazyka je pro nás ovšem dostupná trochu jiná možnost: prohlížeče dokážou zpracovávat kód ve speciálním tvaru **WebAssembly**. Jde o binární formát, který popisuje jednotlivé instrukce virtuálního procesoru, a v porovnání s JavaScriptem je tedy velmi nízkoúrovňový. Jeho výhoda tkví v tom, že do WebAssembly lze kompilovat programy psané v celé řadě jiných jazyků, zejména C, C++ a Rust. V prohlížeči pak nemusí existovat implementace těchto jazyků – převod do WebAssembly musí provést vývojář bokem před tím, než vzniklý kód ke stránce připojí.

Použitím WebAssembly se nám otevírají dvě hlavní nové možnosti:

- spuštění existujícího kódu, který je psaný v jiném jazyce, v rámci webové stránky;

- vytvoření aplikace či knihovny, kterou vykoná prohlížeč bez nutnosti parsování a vykonávání JavaScriptu (vyšší výkon, předvídatelné chování napříč prohlížeči).

U velkých webových aplikací není neobvyklé, když některé jejich součásti tvoří JavaScript (zejména ty, které pracují s uživatelským rozhraním) a jiné zase Web Assembly (třeba ty, které provádí komplikované operace nad velkým množstvím dat). Je ovšem dobré poznamenat, že práce s WebAssembly je komplikovaná a vyplatí se jen tam, kde potřebujeme maximální výkon nebo musíme spolupracovat s existujícím ne-JavaScriptovým kódem. Když se chceme touto cestou vydat, čekají nás především tyto kroky:

1. Zvolit vhodný zdrojový programovací jazyk. Musí k němu existovat takový kompilátor, který dokáže vygenerovat výstup v jazyce WebAssembly, tj. soubory s příponou **wasm**.
2. V rámci běžného klientského JavaScriptu načíst vzniklé **wasm** soubory (typicky pomocí **fetch**).
3. Dodat potřebný podpůrný kód (tzv. *glue code*), který poskytne přemostění mezi logikou WebAssembly a světem klientského JavaScriptu. Pokud například ve WebAssembly chceme načítat data (po síti nebo ze souborů), musíme tuto logiku dodat formou běžných JavaScriptových funkcí. Naopak, načtené WebAssembly funkce budeme chtít jistě volat z normálního JavaScriptu a předávat jim data uložená v JS proměnných.

V rámci této knihy se na WebAssembly podíváme jen v tom nejjednodušším příkladu. Vytvoříme funkci v jazyce C, zkompilujeme ji do WebAssembly a následně ji zavoláme z JavaScriptu. Pro jazyk C existuje celá řada překladačů; my použijeme **Emccripten**, který převádí C/C++ právě do WebAssembly.

Kód naší knihovny bude nabízet jedinou funkci **my_sqrt**, která implementuje celočíselnou odmocninu (a činí tak voláním funkce **sqrt** ze standardní knihovny **math.h**). Může vypada třeba takto:

```
// my_sqrt.c

#include <math.h>
#include <emscripten.h>
```



```
EMSCRIPTEN_KEEPALIVE
int my_sqrt(int x) {
    return sqrt(x);
}
```

Makro **EMSCRIPTEN_KEEPALIVE** pochází z projektu překladače Emscripten, který musíme na svém operačním systému zprovoznit. Jeho účelem je označení těch funkcí, které chceme do výsledného WebAssembly exportovat a zachovat (chytrý překladač by jinak funkci odstranil, neboť v kódu není nikde volána). Jakmile máme Emscripten nainstalovaný, můžeme s ním tento kód zkompileovat:

```
emcc -O3 --no-entry my_sqrt.c -o my_sqrt.wasm
```

V ideálním případě vznikne soubor **my_sqrt.wasm**, který obsahuje danou funkci v binárním formátu WebAssembly. Přesuneme se do známějšího světa běžného JavaScriptu a tento soubor načteme:

```
let response = await fetch("my_sqrt.wasm");
let ab = await response.arrayBuffer();
```

Dostáváme se k poslednímu kroku celé akce, totiž k propojení WebAssembly a JavaScriptu. Použijeme k tomu metodu **instantiate** z objektu **WebAssembly**:

```
let wasm = await WebAssembly.instantiate(ab, {});
```

Druhý parametr je prázdný. Pokud by náš kód v C potřeboval přístup k některým JS funkcím, museli bychom mu je poskytnout právě v druhém parametru funkce **instantiate**. Exportovaná funkcionalita je pro nás nyní dostupná v proměnné **wasm**:

```
let my_sqrt = wasm.instance.exports.my_sqrt;
console.log(42, my_sqrt(42));
```

Protože jsme použili celočíselnou verzi funkce **sqrt**, nepřekvapí nás, že vypsaná hodnota je šest – výsledek je oříznut na nižší celé číslo. Pro potřeby ukázky se jedná o triviální funkcionalitu. Snadno si ale představíme, že místo funkce **my_sqrt** můžeme exportovat složitou logiku, která provede třeba kódování videa, interakci s neuronovou sítí nebo hledání průchodu velkým grafem při plánování cesty na mapě.

Co se nevešlo

Úlohy z minulých kapitol byly vybrány tak, abychom si jejich prostřednictvím prohlédli a vyzkoušeli různé partie JavaScriptu. Pro vážné zájemce tu ještě zbylo několik témat, která nemají společnou úlohu a do celkové architektury knihy tak úplně nezapadají. Abychom zvědavého čtenáře o tyto doplňkové zajímavosti nepřipravili, podíváme se na ně nyní formou neorganizovaného bonusu.

Zelenáči: SVG

V jedenácté kapitole jsme se seznámili s Canvasem. Šlo o kreslicí plochu, se kterou pracujeme pomocí JavaScriptu. Vyzkoušeli jsme různé funkce a zjistili, že canvas je vskutku univerzální nástroj pro rozličné grafické manipulace. Není ale jediný. Ve světě počítačové grafiky typicky uvažujeme dva směry práce s obrazovými daty – rastrový a vektorový. Canvas představuje ten první; prohlížeč nám ovšem dává možnost používat i přístup vektorový. Ten je realizován grafickým formátem SVG (Scalable Vector Graphics).

Do začátku si ujasněme, že obrazový formát je v prohlížeči zpravidla implementován jako podpora pro soubory s danou příponou. Pokud tedy získáme obrazová data uložená v souboru **obrazek.svg** (typicky prostřednictvím specializovaných editorů, jako např. Adobe Illustrator či Inkscape), snadno je můžeme ve stránce zobrazit pomocí HTML:

```

```

Takové řešení je funkční, ale slabé. My bychom rádi pomocí JavaScriptu obsah souboru prozkoumávali, vytvářeli a měnili. *Dovnitř* HTML značky **** ale rozumným způsobem přistupovat nemůžeme. Namísto toho můžeme využít tzv. *inline SVG*, tedy vložení součástí SVG obrázku přímo do HTML. To je možné, neboť SVG jako takové je dialekt jazyka XML – stejně jako HTML používá uzly, atributy a také rozhraní DOM.

Co tedy nalezneme uvnitř typického souboru SVG? Je textový a velmi přímočarý. Ten nejjednodušší může vypadat třeba takto:

```
<svg>
  <circle cx="50" cy="50" r="50" fill="pink" />
</svg>
```

Obalovací značka `<svg>` říká, že nás čeká vektorová grafika. Většinou v ní najdeme informace o rozměrech, resp. souřadnicích, ale ty jsou volitelné. Uvnitř `<svg>` pak pracujeme s tzv. **grafickými primitivy** (pozor – jednotné číslo je zde *grafické primitivum*, nikoliv *grafický primitiv*), která představují základní geometrické tvary a jejich vlastnosti.

K prvkům ze světa SVG můžeme JavaScriptem přistupovat pomocí rozhraní DOM, obdobně jako k ostatním HTML značkám. Můžeme si tak v této podkapitole zkusit znovu implementovat úlohu s favikonkou z jedenácté kapitoly, ovšem tentokrát pomocí SVG. Naším cílem bude vytvořit útvar, který znázorňuje v čase narůstající kruhovou výseč. Nejprve si k tomu připravíme HTML kostru:

```
<head><link rel="icon" /></head>
<svg width="64" height="64" viewBox="0 0 20 20">
  <circle cx="10" cy="10" r="10" fill="lime" />
  <path fill="green" />
</svg>
```

Značka `<path>` představuje naše hlavní grafické primitivum. Jedná se o obecný geometrický prvek, jehož tvar je určen posloupností čar a křivek. Zatím jsme pro výseč nastavili barvu výplně (porovnejme s canvasem, kde jsme barvu museli definovat pomocí JS volání). U rodičovské značky `<svg>` se objevilo několik atributů:

- **width** a **height** říkají, kolik pixelů náš obrázek ve stránce zabere. Pro naši úlohu to nejsou zajímavá čísla, neboť obrázek ve stránce vůbec nepotřebujeme – chceme ho zobrazovat jako favikonku (jejíž rozměry určuje prohlížeč). Pro potřeby ladění ale může být užitečné vidět, co vlastně vytváříme. Tehdy se použijí určené rozměry.

- **viewBox** je jeden z nejdůležitějších atributů SVG. Pomocí něj říkáme, jaké jednotky se budou používat u jednotlivých grafických primitiv. Hodnotou **0 0 20 20** definujeme, že levý horní roh obrázku odpovídá *logickým souřadnicím* 0,0, zatímco šířka a výška prostoru obrázku je *20 logických bodů*. Logické jednotky jsou bezrozměrné a dovolují nám definovat geometrické prvky nezávisle na výsledné velikosti obrázku. Když budeme následně u SVG prvků například uvažovat *logickou souřadnici* 10,10, myslíme tím přesně střed obrazové plochy, nezávisle na tom, kolik pixelů (atributy **width**, **height**) obrázek zabírá.

Hlavní JavaScriptový kód, který v čase mění úhel výseče, můžeme jednoduše zkopírovat z jedenácté kapitoly:

```
let angle = 0;
setInterval(() => {
  angle = (angle + 0.2) % (Math.PI * 2);
  draw(angle);
}, 100);
```

Stačí tedy dodat funkci **draw**, která pro zadaný úhel upraví velikost výseče (a z našeho SVG pak vyrobí favikonku).

SVG značka **<path>** definuje svůj tvar pomocí nezvyklé syntaxe, která – stejně jako u canvasu – odpovídá tažení virtuálním štětcem po plátně. Jedná se o posloupnost písmen (která značí různé druhy posunu štětce) a čísel (která upřesňují souřadnice štětce). Kruhovátka výseč bude sestávat z těchto tří kroků:

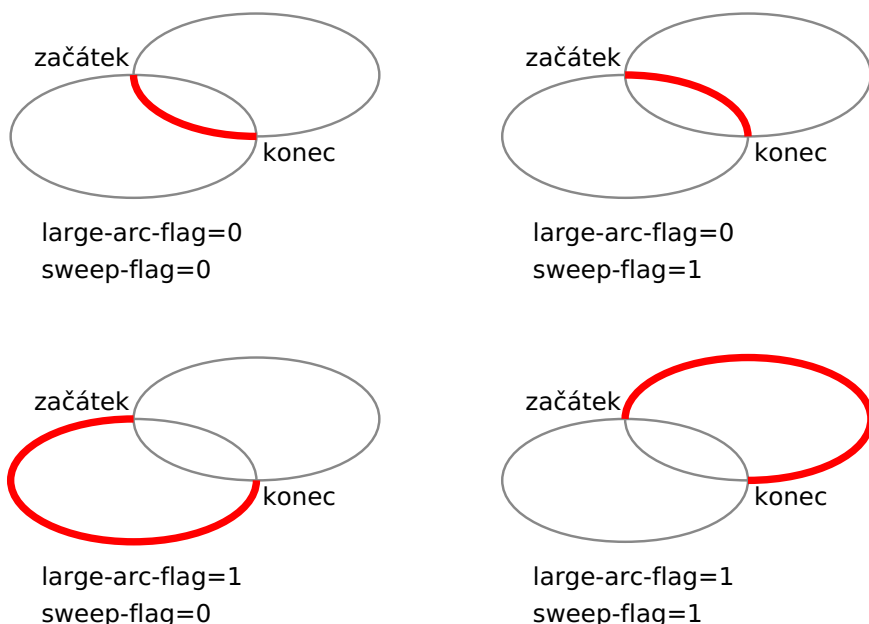
1. **M 10 10**: posun štětce na *logickou souřadnici* 10,10, tedy do středu obrázku
2. **L 20 10**: tah štětce na *logickou souřadnici* 10,10, tedy zcela doprava
3. **A 10 10 0 ? 1 ? ?**: tah štětce po oblouku (části obvodu elipsy)

Otazníky v poslední části trasy značí místa, která musíme dopočítat. Konkrétně pro pokyn **A** (z anglického **arc**, oblouk) potřebujeme:

- velikosti dvou poloos elipsy (v našem případě jde o kružnici a obě jsou 10)
- míru natočení elipsy v ose X (v našem případě 0)
- tzv. *large-arc-flag*, označení toho, zdali chceme zdrojový a cílový bod propojit větším či menším ze dvou možných oblouků

- tzv. *sweep-flag*, označení toho, zdali chceme zdrojový a cílový bod propojit obloukem po směru či proti směru hodinových ručiček
- souřadnice cílového bodu eliptického oblouku

Pokud máme zdrojový a cílový bod, lze je pro zadanou elipsu propojit čtyřmi různými oblouky. Následující obrázek vysvětluje, jak se pomocí dvou příznaků rozliší, který z oblouků máme na mysli:



Obrázek: Dva příznaky upřesňují, který ze čtyř oblouků použít

Ve funkci **draw** cílové souřadnice vypočítáme snadno pomocí goniometrických funkcí. Výsledný tah štětce pak přiřadíme do atributu **d** SVG značky **<path>**:

```
function draw(angle) {
  let svg = document.querySelector("svg");

  let x = 10 + Math.cos(angle) * 10;
  let y = 10 + Math.sin(angle) * 10;
  let largeArc = (angle > Math.PI ? 1 : 0);
  let d = `M 10 10 L 20 10 A 10 10 0 ${largeArc} 1 ${x} ${y}`;
```

```
svg.querySelector("path").setAttribute("d", d);
}
```

Posledním krokem je vytvoření favikonky z tohoto obrázku. U canvasu z jedenácté kapitoly jsme její URL získali metodou **toDataURL** a jednalo se o velice dlouhý řetězec, který obsahoval informace o barvě všech pixelů zpracovávaného plátna. Při použití SVG stačí zadaný DOM prvek prostě převést na řetězec a z něj vytvořit *data URI*. K tomu můžeme použít vestavěný objekt **XMLSerializer**:

```
let link = document.head.querySelector("[rel=icon]");
let str = new XMLSerializer().serializeToString(svg);
link.href = `data:image/svg+xml,${str}`;
```

Tyto řádky umístíme na konec funkce **draw**, stejně jako tomu bylo v případě generování favikonky z canvasu.

Koumáci: striktní režim

Tato podkapitola je přítomna hlavně pro úplnost. Při obhlížení některých užitečných konceptů JavaScriptu (moduly v druhé kapitole, klíčové slovo **this** v osmé kapitole) jsme na zmínku o striktním režimu narazili a byla by škoda toto téma nevysvětlit, byť v praxi mu rozumět nemusíme.

Myšlenka striktního režimu se objevila v roce 2009. JavaScript zažíval velký boom a vývojáři uvažovali, zda lze jazyk vylepšovat nejen přidáváním nových vlastností, ale i změnou těch existujících. Klíčovým problémem byla samozřejmě zpětná kompatibilita, která je na webu důležitým tématem. Ve verzi ES5 se proto objevila možnost použití striktního režimu, který mění některé problematické prvky jazyka a zároveň je dostupný formou *opt-in*, tj. jen pro ty části kódu, kde si to programátor explicitně vyžádá.

Volitelné zapnutí striktního režimu se provádí přidáním řetězce **"use strict"**; na první řádek kódu. Pokud tak učiníme ve funkci, bude se striktní režim aplikovat pouze na ni. Pokud zmiňovaný řetězec dáme na začátek souboru, bude se striktní režim aplikovat na celý soubor.

Striktní režim se následně ukázal jako dobrý nápad, takže jeho aktivace je dnes ještě jednodušší. Pokud svůj kód napíšeme jako ES modul (použijeme direktivy `import`, `export` nebo HTML atribut `type="module"`), bude automaticky vykonán ve striktním režimu. Pojdme se tedy podívat, jak že se striktní režim liší od toho historického (označovaného jako *sloppy mode*, tj. něco jako *lajdácký*). Zde je výčet nejdůležitějších změn:

- **Nutnost definice proměnné pomocí klíčového slova.** Bez striktního režimu je možné definovat proměnnou prostým zápisem `a = 42`, což způsobí vznik či přepsání globální proměnné.
- **Vyvolání výjimky při přiřazení do některých globálních proměnných.** Bez striktního režimu se můžeme pokusit přiřadit např. do proměnné `undefined`, `NaN` či `Infinity`. Tato operace selže a my se o tom nedozvíme. Ve striktním režimu taková operace způsobí výjimku.
- **Nula na začátku čísel.** Historicky se nula na začátku používala k zápisu čísel v osmičkové soustavě (podobně jako prefix `0x` značí soustavu šestnáctkovou). V praxi taková funkce vývojáře spíš mátlá, takže ve striktním režimu způsobí nula na začátku výjimku.
- **Odstranění klíčového slova `with`.** To umožňovalo expanzi vlastností objektu do lokálních proměnných – v praxi sice užitečné, ale velmi nepředvídatelné chování. Ve striktním režimu zakázáno.
- **Použití `this` ve funkci volané obyčejně, bez tečky.** Takový zápis není k ničemu užitečný a proto ve striktním režimu při tomto volání `this` nabývá hodnoty `undefined`.
- **Odstranění vlastností `caller` a `callee`.** Pomocí nich bylo možné v těle funkce poznat, z jaké jiné funkce je ta současná volána. To má negativní dopad na výkon a bezpečnost, takže ve striktním režimu již tyto vlastnosti nejsou dostupné.
- **Nová klíčová slova,** která nesmíme použít jako názvy vlastních proměnných (např. `let`, `interface`, `implements` a další). Vzniká tak prostor pro budoucí rozšiřování syntaxe jazyka.

Profíci: iterační protokol

Ve třetí kapitole jsme se seznamovali s různými formami iterace. Při použití syntaxe **for-of** dochází k **programovatelné iteraci**, kdy procházený objekt sám specifikuje, co to znamená *vrátit další položku*.

Představme si značný objem textu, který potřebujeme zpracovávat po řádcích. Může jít o velký dokument získaný prostřednictvím HTTP požadavku, logovací soubor nebo cokoliv jiného. Rozdělit text na jednotlivé řádky je triviální (stačí použít metodu **split**, které předáme rozdělovací podřetězec), ale zároveň nešikovné s ohledem na výkon. Při převodu řetězce na pole vznikne fakticky duplikát původního textu, takže touto operací zabere alespoň dvojnásobek paměti. Chytřejší je v původním textu postupně vyhledávat oddělovače řádků a soustředit se na jednotlivé podřetězce mezi nimi.

Toto chytřejší procházení můžeme vytvořit jako iterovatelný objekt. Zadáme mu velký vstupní text a oddělovač; pak mu budeme opakovaně říkat „*dej další kousek!*“ tak dlouho, než dojdeme na konec procházených dat.

Přesně takhle funguje iterační protokol. Musíme vytvořit objekt s metodou **next()**, která vrátí další iterovanou položku a také příznak, zdali už jsme na konci, nebo můžeme pokračovat. Jeho návratovou hodnotou proto bude objekt s vlastnostmi **value** (vrácená hodnota) a **done** (bool, jsme-li na konci). Proces iterace je stavový (mezi jednotlivými voláními **next()** si musíme pamatovat, kde jsme minule oddělovač našli), takže pro jeho realizaci často používáme uzavěru:

```
function createIterator(text) {
  let separator = "\n";
  let lastIndex = 0;
  return {
    next() {
      if (lastIndex > text.length) {
        return { value: undefined, done: true };
      }

      let index = text.indexOf(separator, lastIndex);
      if (index == -1) { // oddělovač nenalezen
        let value = text.slice(lastIndex);
```



```

        lastIndex = text.length+1;
        return { value, done: false };
    } else {          // oddělovač nalezen
        let value = text.slice(lastIndex, index);
        lastIndex = index + separator.length;
        return { value, done: false };
    }
}
}
}
}

```

Při hledání oddělovače využíváme metodu **indexOf** a zejména její volitelný druhý parametr, který říká, od kolikátého znaku hledáme. V proměnné **lastIndex** si pamatujeme, kde jsme minule hledání skončili. Při zavolání **next()** se tak můžou stát tři různé věci:

1. Po-

kud jsme se minulým hledáním dostali na konec textu, vrátíme v odpovědi **done: true** (hodnota není relevantní, náš iterátor již nic nevrací).

2. Pokud jsme v neprozkoumané části textu našli oddělovač, vrátíme podřetězec (od minulého oddělovače k novému) a zapamatujeme si pozici konce tohoto nového oddělovače.

3. Pokud jsme už žádný další oddělovač nenašli, vrátíme zbytek textu (od minulého oddělovače do konce).

Z toho plyne, že i když v textu žádný oddělovač nenalezneme, metoda **next()** vždy vrátí alespoň jednu hodnotu (poslední položka v seznamu výše).

Abychom tento objekt (když má metodu **next()**, říkáme mu *iterátor*) mohli při iteraci použít, zbývá ho vložit na správné předem dohodnuté místo. Konkrétně jej musí vrátit funkce, která je v iterovaném datovém typu dostupná pod klíčem, jehož hodnota je **Symbol.iterator**. Zní to složitě, ale implementace je jednoduchá. Můžeme jí docílit například třídou:

```

class LineIterator {
  constructor(text) {
    this.text = text;
  }
}

```

```
[Symbol.iterator]() {  
    return createIterator(this.text);  
}
```

Tím je obřad tvorby iterátoru dokončen a my jej můžeme s velkou slávou použít:

```
let data = "a\nb\nc";  
let iterator = new LineIterator(data);  
for (let line of iterator) console.log(line);
```

Mimochodem: výše uvedený kód implementuje tzv. *synchronní iterační protokol*. V praxi se můžeme setkat s variantou, kdy je funkce **next()** asynchronní (tj. vrací Promise, protože například stahuje data HTTP požadavkem a podobně). Takový asynchronní iterátor pak umístíme pod klíč **Symbol.asyncIterator** a pracujeme s ním syntaxí **for await (let x of ...)**.