Tvorba webových aplikací

přednáška

Michal Bubílek
Varnsdorf 10/2019

Anotace

Výukový materiál se zabývá tvorbou webových aplikací se zaměřením nejen na serverovou část. Student se postupně seznamuje se všemi zásadními tématy od webové komunikace, kódování webových stránek, přes objektové programování až k programování na straně serveru a komunikace s databázovými servery. Objektové programování v PHP na straně webového serveru je stěžejní částí výukových materiálů. Student se však dostatečně seznamuje i s technologiemi HTML a CSS, které jsou právě na serveru jazykem PHP obvykle generovány. Materiál není koncipován jako kompletní referenční příručka jazyků, nýbrž čtenáře seznamuje s problematikou a typickými modelovými příklady a nabízí a vysvětlí řešení. Budou zde sepsány klíčové vlastnosti jazyků, které budou vysvětleny a názorně ukázány na příkladech.

Klíčová slova

OOP, web, PHP, HTML, CSS, praktické programování

Obsah

1 Webové aplikace

Název těchto učebních materiálů je „Objektové programování pro web“. Řekněme si něco o jednotlivých slovech tohoto názvu. Programování je činnost, při které vzniká software (programové vybavení) a pokud užitečně slouží uživateli pro jakýkoliv obor jeho činnosti, tak tomu říkáme aplikace (aplikační software). Pokud jsou tyto aplikace v prostředí webu, říkáme jim webové aplikace. V těchto studijních materiálech se tedy budeme zabývat tvorbou webových aplikací dle **OOP** (***Objektově orientované programování***). O objektovém programování si řekneme později. Nyní se musíme nejprve seznámit se základními pojmy, principy webové komunikace, zásady tvorby webových aplikací…

Webová aplikace je tedy aplikační programové vybavení běžící v prostředí webu. **Web** (***World wide web, www, v překladu „světová rozsáhlá síť“***) je ona „celosvětová pavučina“, což chápeme jako systém na celosvětové síti Internet, který se stará o prohlížení, ukládání a odkazování dokumentů. Celý tento systém dokumentů tvoří hypertext, což je nelineárně strukturovaný text obsahující odkazy (hyperlinky) na své vlastní části. Běžně jsou tyto dokumenty známy jako webové stránky. Za posledních dvacet let se však díky pokroku v moderních technologiích webové stránky velmi změnily. Nyní se obvykle již nejedná o statický dokument (webovou stránku), ale již právě o plně interaktivní webovou aplikaci, která je mnohdy k nerozeznání od běžné desktopové aplikace. A to, že se jedná o webovou aplikaci, poznáme jen dle toho, že je spuštěna ve webovém prohlížeči (internetovém prohlížeči, webovém browseru). Příkladem nám mohou být moderní sociální sítě, webové obchody (elektronické obchody, eshopy), cloudové služby, služby vyhledávání dopravního spojení, emailoví klienti, hry, dokonce i běžný kancelářský software v podobě textového editoru, tabulkového procesoru a mnoho dalšího.

V předcházejícím odstavci, který nám osvětloval, co jsou webové aplikace, bylo použito mnoho termínů, které byly v kontextu odstavce vysvětleny jen velmi minimalisticky nebo vůbec. Nyní si uděláme jednoduchý přehled základních pojmů, které nás budou doprovázet v průběhu celého tohoto studijního materiálu.

Internet je celosvětová počítačová síť (síť sítí). Přesněji můžeme říci, že je to celý systém propojených počítačových sítí. Tento systém běží díky rodině protokolů **TCP/IP** (***Transmission Control Protocol/Internet Protocol – „přenosový protokol transportní vrstvy/protokol síťové vrstvy“***), což není nic jiného než soubor pravidel pro komunikaci mezi jednotlivými uzly sítě na všech logických vrstvách. Internet je využíván pro mnoho služeb. Typickým příkladem je právě služba www nebo elektronická pošta (emaily). Počátky internetu sahají až do konců šedesátých let minulého století, kdy vznikla síť ARPANET, která byla financována ministerstvem obrany Spojených států amerických a měla za úkol zajistit decentralizované propojení počítačů.

Web je informační systém (služba) v síti internet, díky které můžeme prohlížet, ukládat a odkazovat na dokumenty v rámci celého světa (soustava propojených hypertextových

dokumentů). Web vynalezl v roce 1989 sir Tim Berners-Lee. Web je postaven na třech základních pilířích, kterými jsou protokol **HTTP** (***Hypertext Transfer Protocol***), značkovací jazyk **HTML** (***Hypertext Markup Language***) a lokátor zdrojů **URL** (***Uniform Resource Locator***).

W3C (***World Wide Web Consortium***) je mezinárodní konsorcium (sdružení lidí) založené za účelem vývoje webových standardů. Zabývá se například přístupností webů, vzděláváním, standardy HTML, **XHTML** (***eXtensible Hypertext Markup Language***), **CSS** (***Cascading Style Sheets – Kaskádové styly***), **XML** (***Extensible Markup Language***), **SVG** (***Scalable Vector Graphics***), **DOM** (***Document Object Model – Objektový model dokumentu***) a dalšími. Konsorcium založil roku 1994 Tim Berners-Lee, který doposud konsorciu předsedá a je často označován jako „otec webu". Za „otce internetu" je označován americký informatik Vint Cerf, který spolu s Bobem Kahnem vytvořili komunikační protokoly TCP/IP.

Často je pod pojmem web chápána webová stránka a mezi uživateli této služby je to velmi rozšířené. To nepokládám za špatné a sám to budu v těchto materiálech používat například při vysvětlování pojmu „dynamický web", kde tím myslím dynamickou webovou stránku a nikoliv celou službu v síti internet. Věřím, že to nebude matoucí a každý z kontextu hned pochopí, co je termínem web právě myšleno.

Hypertext je nelineárně strukturovaný text obsahující odkazy na své vlastní části. Nemá tedy začátek ani konec. Hypertext tvoří obsah webu (text, obrázky, video, audio…), ke kterému máme přístup díky protokolu HTTP a vidět jej můžeme obvykle v podobě webových stránek.

HTTP a **HTTPS** jsou protokoly z rodiny protokolů TCP/IP, které se starají o komunikaci mezi webovým klientem a webovým serverem. Ze samotného názvu je patrné, že se jedná o protokoly určené pro přenos hypertextu. U protokolu HTTPS je oproti HTTP komunikace šifrovaná pomocí **TLS** (***Transport Layer Security***) nebo **SSL** (***Secure Sockets Layer***). Zabezpečená verze protokolu je doporučovaná nejen pro webové aplikace, kde se internetem mezi klientem a serverem mohou přenášet citlivá data, ale i pro běžné webové stránky. Hostingové firmy nabízejí běžně certifikáty bezplatně. Protokol HTTP používá obvykle port TCP/80 a HTTPS pak TCP/443.

URL je textová adresa jakéhokoliv zdroje dat v prostředí webu. URL adresou tak můžeme určit nejen webovou stránku, ale i obrázky, soubory kaskádových stylů, java skriptů a veškerý další webový obsah. Adresa se skládá z několika povinných a nepovinných částí, které mají jasně stanovená místa.

Jako příklad si uveďme: https://www.example.com:80/app/index.php?p=34&l=cz. Na začátku je uveden přenosový protokol, který pokud se neuvede, bude standardně HTTP. Poté následuje povinná doménová adresa serveru, která může být zapsána i v podobě číselné IP adresy. Za **TLD** (***Top Level Domain***) .cz je uvedeno číslo portu 80. Pokud se neuvede, tak je právě 80 standardem, na kterém poslouchá webový server. Za portem pak může být uveden přímo požadovaný soubor včetně adresářů. Pokud se neuvede,

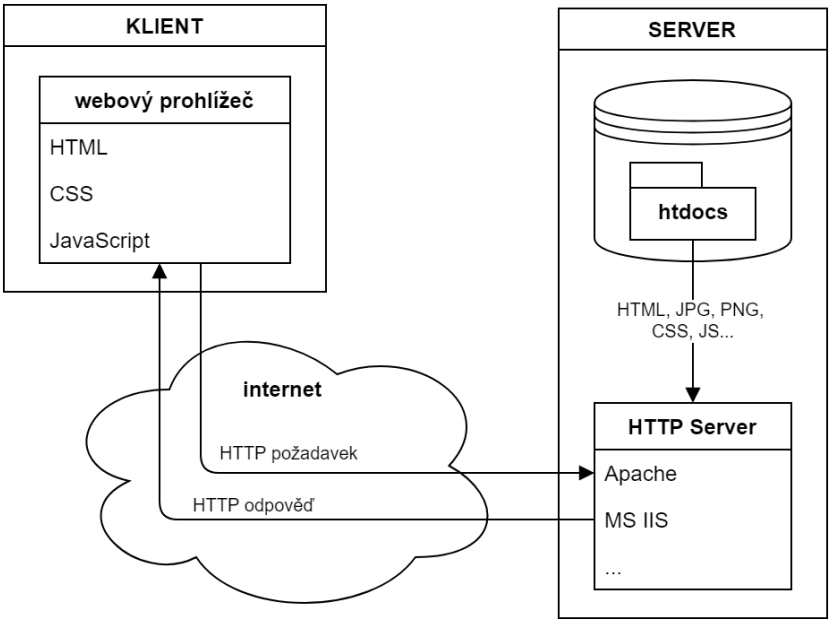
webový server hledá ve výchozím adresáři obvykle soubory index.html, index.php nebo default.html atp. Jako poslední jsou za otazníkem předávány GET parametry „p“ a „l“, které jsou odděleny „&“ (et, ampersand). Výše uvedený příklad URL adresy není však dobře zapamatovatelný a může vypadat klidně i komplikovaněji, a proto je jistě uživatelsky přívětivější používání tzv. COOL URL adres. Taková adresa by pak vypadala například: https://www.example.com/uloha-na-url. Takováto adresa je vhodná i pro internetové boty (crawler, spider, gatherer), kteří web navštěvují a indexují. Do databází se tak URL adresa dostává i s klíčovými slovy.

Webový server (Webserver, HTTP server) je označení pro počítač, který nabízí webové služby. Webserver vyhovuje HTTP požadavkům nespočtu webových klientů. Od klienta přijme HTTP požadavek, a pokud je to možné, pošle cíl specifikovaný URL adresou uvedenou v požadavku. Jako webserver bývá označován i počítačový program, který na serveru plní výše popsanou roli. Typickými příklady takovýchto software jsou Apache HTTP Server a Microsoft IIS (Internet Information Services). Pro podporu webových aplikací běží obvykle na webových serverech i **SŘBD** (***Systém řízení báze dat***), které známe v angličtině jako **DBMS** (***Database Management System***) a programový interpret. Nejčastějšími představiteli SŘBD jsou dle mé zkušenosti MySQL, MariaDB, MS SQL Server a nejčastějšími programovými jazyky pak **PHP** (***rekurzivní zkratka PHP: Hypertext Preprocessor***), Python, ASP a Java.

Webový klient využívá pomocí protokolu HTTP služeb webového serveru. Nejčastějšími příklady jsou webové prohlížeče (browser). Typickými konkrétními představiteli jsou pak Google Chrome (Blink), Mozilla Firefox (Gecko), Internet Explorer (Trident), Microsoft Edge (EdgeHTML/ Chromium) a Opera (Blink). V závorce jsou uvedená zobrazovací jádra, na kterých jednotlivé prohlížeče běží. Webový browser zobrazuje graficky nebo textově obsah webu. Standardně musí každý browser zvládnout technologie HTML, JavaScript a CSS. Dají se rozšířit o zásuvné moduly podporující Java applety, Flash animace…

1.1 Principy webové komunikace

Nejprve si vysvětlíme principy webové komunikace pro statický web, který je zobrazen na obrázku.

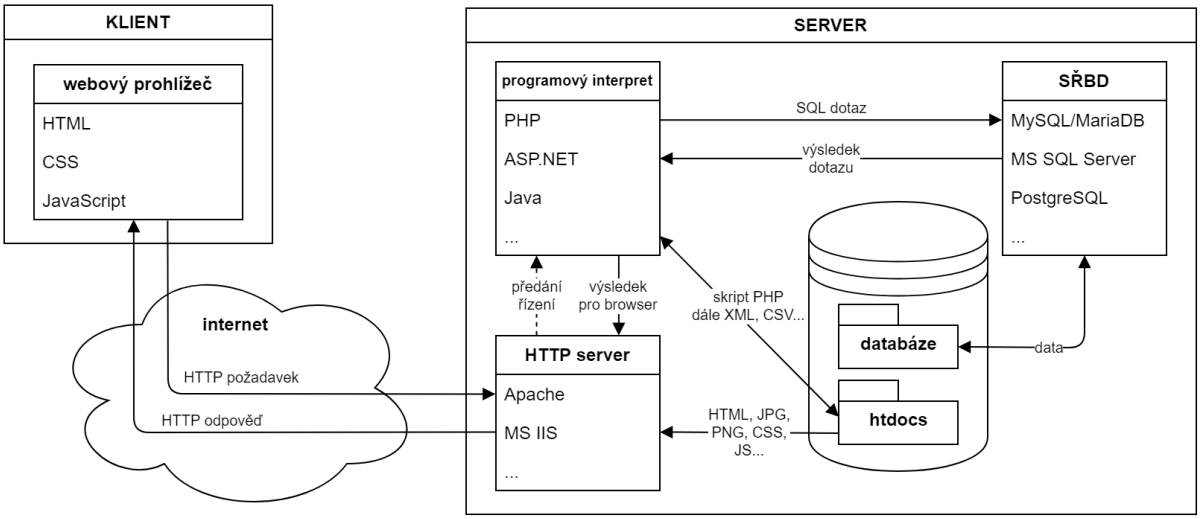


Obrázek 1 Schéma webové komunikace statického webového obsahu

Uživatel zadá v okně prohlížeče patřičnou URL adresu webového zdroje (stránka, obrázek, CSS, **JS** (***JavaScript***)…). Webový klient tak vyšle do internetu HTTP požadavek. Ve skutečnosti se na klientovi z pohledu počítačových sítí děje mnoho dalších věcí jako například zapouzdřování (encapsulation) HTTP požadavku do TCP segmentu, ten do IP paketu, ten do Ethernetového nebo Wi-Fi rámce, který se již v podobě nul a jedniček vysílá bit po bitu na přenosové medium. Také se pravděpodobně využije služba **DNS** (***Domain Name System***), abychom zjistili IP adresu cílového webového serveru, kterou potřebujeme do hlavičky IP paketu. Tyto detaily jsou však pro jiný studijní materiál týkající se spíše počítačových sítí. HTTP požadavek doputuje internetem až na webový server. HTTP server, nyní míním software, rozhodne zejména dle URL, co se bude dít. Uvádím „zejména“, protože v rozhodování serveru hrají roli i informace v hlavičce HTTP dotazu. V případě, že se klient dožaduje jen souborů html, css, js, pdf, jpg, png, svg a podobný statický obsah, samotný HTTP server najde soubor na disku dle své nastavené bezpečnostní politiky a odešle jej v HTTP odpovědi klientovi. Každá takováto HTTP odpověď má svůj číselný kód. V případě, že je vše v pořádku a soubor se odesílá klientovi, kód je 200. Dále pak se můžeme často setkat s kódy 301 (přesměrováno), 400 (špatně zadaný dotaz), 401 (neautorizovaný přístup), 403 (zakázaný přístup), 404 (soubor nenalezen) a 500 (chyba serveru). Klient přijme HTTP odpověď se zaslaným souborem ze serveru a patřičně jej zobrazí, nebo pokud to nejde, nabídne jej ke stažení.

Další obrázek představuje komunikaci pro dynamické webové stránky. Je již komplikovanější, ale jen na straně serveru, klient je stejný jako u statického webu. Stanovme si pojem „dynamický web“. Mnozí tento pojem vysvětlují tak, že se jedná o webovou stránku, která obsahuje mnoho dynamických prvků. Příkladem může být karusel s obrázky, rozbalovací menu, animace a jiné aktivní prvky. Neobsahuje tedy jen statický text. My se na to budeme však dívat jinou optikou. Vše, co se děje na straně klienta, pro nás nebude podstatné. Podstatné je pro nás chování na webovém serveru. Dynamický web budeme

chápat jako takový, jehož obsah se na serveru generuje programem. Samotná stránka je tedy sestavována až při jejím samotném volání, před tím na serveru neexistovala. Samozřejmě mohou existovat různé kešovací systémy, ty ale nyní pomineme, protože to nijak nenaruší logiku věci. Vlastně se jen kešuje (dočasně ukládá) dříve programem vygenerovaný obsah. Důvodem je zkrácení doby sestavování stránky na serveru.



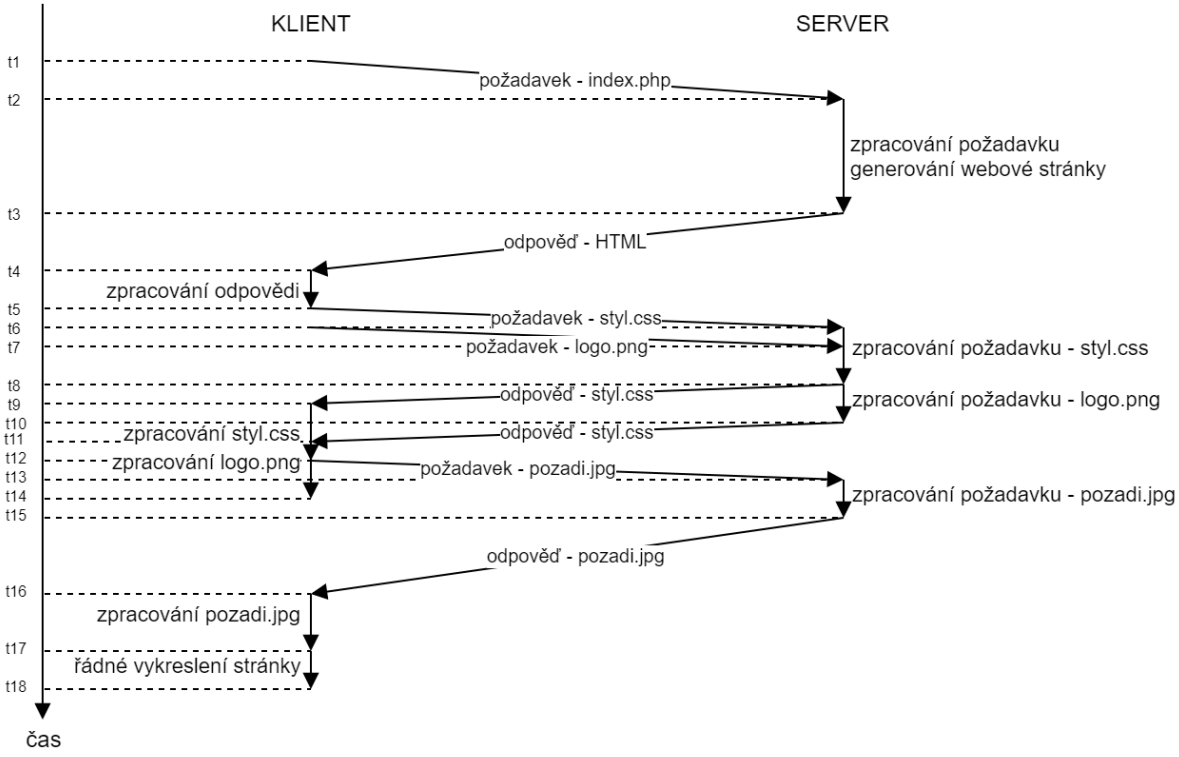
Obrázek 2 Schéma webové komunikace dynamického webového obsahu

Na schématu můžeme vidět, že se na straně klienta nemění nic. Klient tedy opět pošle HTTP požadavek na server. Server tento požadavek přijme a v případě, že vyřízení neznamená jen najít patřičný obsah na disku, jej předá patřičnému programovému interpretu. Programový interpret si najde na disku zdrojový kód a ten řádek po řádku vykoná. Ve zdrojovém kódu mohou být příkazy na práci s databází, komunikaci s dalšími službami na různých serverech v celém prostředí webu a mnohé další. Výsledkem takového programu je obvykle HTML kód webové stránky. Může tím být ale jakýkoliv jiný soubor. Často se tak může generovat krom html i pdf, jpg, png, css, js, xml, csv… Vygenerovaný soubor se předá HTTP serveru, který jej pošle jako jakýkoliv jiný webový obsah webovému klientu. Webový klient tak na stejný dotaz může obdržet vždy jiný, aktualizovaný, obsah. V tomto chápejme onu dynamičnost webu. Běžně se s tímto setkáváme například u internetových obchodů, informačních novinových portálů, sociálních sítí a dalších webech, kde se obsahy stránek mění i v řádu sekund. Podobně i u webových služeb, které generují obsah dle našich specifických zadání. Příkladem jsou pak různé vyhledavače, jízdní řády, rezervační systémy a mnohé další.

Často se setkávám s nepochopením této koncepce a studenti vkládají například PHP zdrojový kód přímo do browseru, který jej samozřejmě neprovede. Tento zdrojový kód musí projít PHP interpretem na straně serveru, který teprve pro browser jeho obsah sestaví. To vše lze provozovat, a často se i ve vývojovém prostředí provozuje, na jednom počítači.

Nyní se pojďme podívat na zpracování dynamické webové stránky z pohledu času, kde jsou znázorněny časové prodlevy od poslání požadavku až po zobrazení webové strán-

ky. Jedná se o jednoduchý web s obrázkem loga a obrázkem pozadí, který je definován v externím css souboru.



Obrázek 3 Časová osa zpracování webové stránky

V čase t1 pošle klient webovému serveru požadavek na webovou stránku index.php. V čase t2 server požadavek obdrží a začne jej zpracovávat. Rozhodne o tom, že se jedná o dynamický web a předá řízení programovému interpretu PHP, který kód stránky v jazyce HTML vygeneruje. Až v čase t3 webový server odpovídá a posílá výsledný vygenerovaný HTML kód požadované webové stránky a ten v čase t4 klient obdrží. Prodleva mezi t1 – t4 se nazývá **TTFB** (***Time to first byte***). Klient začne HTML kód zpracovávat. Již v hlavičce obdrženého HTML kódu je odkaz na CSS soubor, ve kterém je definován vzhled webové stránky. Klient tedy pošle v čase t5 serveru druhý požadavek, tentokráte na styl.css. V čase t6 server tento požadavek obdrží a začne jej zpracovávat. Zároveň klient přišel na to, že bude potřebovat i obrázek logo.png a vyšle tedy již třetí požadavek na server o obrázek loga, který server obdrží v čase t7. Server tyto požadavky vyřídí a v časech t8 a t10 zasílá patřičné soubory css a jpg klientovi. Ten je v časech t9 a v t11 obdrží. V čase t12 klient po zpracování designového souboru styl.css zjistí, že potřebuje obrázek pozadi.jpg a zažádá o něj. Server požadavek obdrží v čase t13 a v čase t15 již posílá obrázek pozadi.jpg klientovi, který jej v čase t16 obdrží, v čase t17 zpracuje. Celý web je postupně v různých mezičasech již zobrazován. Browser se snaží uživateli web zobrazit co nejdříve. Celý web je kompletně zobrazen až v čase t18.

Nyní poukážeme na kritické části, které mají na dobu načítání stránky vliv. Co jistě neovlivníme, je čas t1-t2, tedy dobu, po kterou půjde požadavek sítí od klienta k serveru. Již ale můžeme ovlivnit to, jak dlouho půjde odpověď od serveru ke klientovi. Jednoduše se

budeme snažit posílat od serveru ke klientovi co nejmenší data. Sice neovlivníme rychlost sítě, ale menší data se budou přenášet rychleji než data objemná. Již ve schématu je vidět, jak výrazně rychleji putuje obrázek malého loga a jak se prodleva zvětší u většího obrázku pozadí. Není dobré, aby obrázky měly velikost v řádu MB. Snažíme se je držet co nejmenší. Používáme nekonečné navazující pozadí, vektorovou grafiku, snažíme se využít různé „finty“, které nám nabízí CSS. Také se snažíme posílat co nejméně požadavků na server, proto používáme obrázkové mapy například pro ikonky, stylové soubory shrneme do jednoho a minifikujeme a podobně. Co je kritické a můžeme ovlivnit, je doba generování obsahu na serveru. Jednak se snažíme programovat tak, abychom neplýtvali zdroji, což je pro nás procesor a paměť. Programy píšeme tak, aby byly nenáročné a generovaly obsah co nejrychleji. Také můžeme mnoho ovlivnit volbou hostingové společnosti. Optimální je, když celková doba t1 – t18 je pod 1–2 sekundy a TTFB je do 250 ms. V opačném případě již není pro návštěvníka webu příjemné se po webu pohybovat a při každém kliknutí do jiné sekce webu další více jak dvě sekundy čekat.

1.2 Zásady tvorby webových aplikací

Povíme si, jakými způsoby se webové aplikace tvoří a na co je dobré myslet při jejich tvorbě, aby dobře fungovaly. Realizace takové aplikace je projekt. Někdy může být menší, jindy zase větší, ale vždy bychom měli myslet na to, že se jedná o projekt a měli bychom brát v úvahu všechny jeho aspekty. Každý projekt má čtyři základní fáze: specifikace, implementace, validace a evoluce.

1.2.1 Specifikace

Výsledkem specifikace musí být jasné zadání pro grafiky, kodéry a programátory. Přechází se od ideje, která je obvykle vágní, ke konkrétním popisům designu a funkcí nové aplikace. Řeší se zde i studie proveditelnosti, ve které se musí řádně zhodnotit, je-li možné projekt realizovat. Překážkami realizace mohou být nedostatečné zkušenosti, nedostatečná znalost technologií, nedostatek lidských zdrojů, času, finančních prostředků, ale třeba i nedostatečně provedená specifikace. I to není stále vše. Pokud se již rozhodneme aplikaci vytvořit, je nutné si na začátku vždy umět odpovědět na několik základních otázek:

1. Jaký je cíl webové aplikace?
2. Jaké jsou cílové skupiny?
3. Jakým způsobem webovou aplikaci vytvořit?
4. Kde provozovat?
5. Kdo bude webovou aplikaci spravovat?
6. Kdo a jak často bude aktualizovat (systém i obsah)?
7. Kolik to bude stát?
8. Jaký je termín spuštění?

Pojďme si přiblížit oněch osm výše uvedených otázek, i když bychom si jistě mohli položit mnoho dalších. Znát dobře cíl aplikace je klíčové. Uvědomme si, že aplikace se nevytváří jedním způsobem a podle toho jaký mají obsah, pak plní patřičnou funkci. Už od

začátku by se měla aplikace navrhovat podle toho, bude-li něco prodávat, nebo fungovat jako virtuální galerie. Rozdíl pak nebude jen v prvcích na webové stránce, uspořádání menu, designu vůbec, ale i v technologiích, které obsah připravují. Znát cílovou skupinu je také velmi důležité. Tvoříme obsah pro lidi nebo boty. Lidé chtějí „user friendly" prostředí, kde zohledňujeme i lidskou psychologii, botům záleží zejména na validním kódu. Cílová skupina pak ovlivňuje i lokalizace obsahu, způsob podání obsahu, odbornost textů a mnohé další.

Další otázkou je, jak se webové aplikace vůbec tvoří? Zde jsou tři základní způsoby, které se mohou různě prolínat. Prvním je psát aplikaci „z čisté louky". To znamená si veškeré zdrojové kódy psát sami. Dalším způsobem je používat různá hotová řešení pro dílčí části. To mohou být různé frameworky, knihovny, redakční systémy, ale i fonty, ikonky, obrázky, zvuky a podobně. Třetím způsobem je využít hotové kompletní řešení jako službu (Google, Facebook, Booking…). Zde nesmíme zapomenout na otázky, zda je možné hotová řešení upravovat a rozšiřovat o nové funkce. Zda jsou hotová řešení svými programátory dále vyvíjena a udržována. Zda je možné tato řešení legálně použít, tedy jsou-li vydávána pod požadovanými licencemi.

Je také důležité, kde a jak budeme webovou aplikaci z technického pohledu provozovat. Opět máme několik možností:

1. Dedikovaný server (Pronajatý fyzický server, kde se poskytovatel stará o HW. Zákazník si řeší kompletně SW.)
2. Managed server (Pronajatý fyzický server, kde se poskytovatel stará o HW i SW. Zákazník má jen přístupy na FTP server a do databáze.)
3. Server housing (Poskytovatel nabízí jen prostor pro server, konektivitu k datové síti, napájení a chlazení. Server a jeho kompletní správu řeší zákazník.)
4. Virtuální server (Zákazník si pronajímá jen části serveru s virtualizací, kde má obvykle vlastní kopie operačního systému a nainstalované aplikace. Poskytovatel se stará o server a chod virtualizace.)
5. Běžný hosting (Zákazník neřeší správu serveru, má jen přístupy k FTP, na email, do databáze a do administrace svého účtu. Poskytovatel se stará o HW i SW na serverech. Na jednom serveru může být i několik stovek nebo tisíců zákazníků.)
6. Vlastní webserver (Zákazník si řeší vlastní server, prostory, chlazení, konektivitu, elektřinu a kompletní software.)

Na každém tomto řešení musíme řešit mnoho dalších záležitostí. Naše výsledná aplikace bude potřebovat patřičný prostor, programový interpret, napojení na databázi a také i konkrétní HTTP server. Nejčastěji se setkáváme s kombinací **LAMP** (***Linux Apache MySQL PHP***), **WAMP** (***Windows Apache MySQL PHP***) nebo **MAMP** (***Mac Apache MySQL PHP***). U Windows řešení najdeme kombinace Windows + IIS + ASP.NET + MS SQL Server. Častou alternativou nebo i souběžným SŘBD k MySQL je nyní i MariaDB, která z MySQL vznikla po koupení MySQL společností Oracle Corporation. Mnohdy nás zajímají i verze jednotlivých aplikací či technologií, protože naší programované aplikaci mohou od určitých verzí něco přinášet.

Kromě hostingu nesmíme zapomenout i na doménu. Volba vhodné domény je velmi důležitá. Měla by být jasná, jednoduchá, krátká a výstižná. Cena domény se pohybuje řádově ve stokorunách ročně, což je v rámci projektu zanedbatelná částka.

Pokud vytvoříme například internetový obchod, musí být jasné, kdo bude aktualizovat jeho obsah. Kdo bude plnit obchod produkty. Bude-li to automatický systém synchronizace z nějakého externího feedu dodavatele, nebo bude-li to dělat nějaký zaměstnanec obchodu ručně, nebo kombinace obou přístupů.

Poslední dvě z oněch osmi základních otázek není již třeba více rozvíjet. Řekněme si jen základní projektové poučky. U řízení projektu existuje tzv. projektový trojimperativ (Triple Constraint nebo the Iron Triangle). Jde o pomyslný trojúhelník, kde na jeho vrcholech je funkce, cena a termín. Tyto tři prvky se velmi vzájemně ovlivňují. Například pokud zkrátíme termín, tak se to jistě negativně projeví na ceně nebo funkcích, nebo obojím. Podobně pokud přidáme nové funkce, odrazí se to negativně na čase nebo ceně, nebo opět obojím. Analogicky se podobná negativa dějí u zkrácení ceny. Na tyto závislosti nesmíme zapomenout. Je dobré si jeden z těchto parametrů projektu vzít jako stěžejní a zbylé dva k němu poté přizpůsobovat. Vše, zejména pak termíny, by mělo být realistické.

1.2.2 Implementace

Ve fázi implementace již tvoříme aplikaci dle zadání. Hledáme řešení konkrétních dílčích problémů. Zapisujeme algoritmy v programovacích jazycích. Provádíme testování napsaných částí aplikace.

Každý programátor má své oblíbené vývojové prostředí, které známe pod zkratkou **IDE (Integrated Development Environment)**. Můžeme uvést například NetBeans, Eclipse, KDevelop, Microsoft Visual Studio nebo i jednoduchý, ale velmi užitečný PSPad. Tato prostředí krom zvýraznění částí zdrojového kódu nabízí mnoho dalších užitečných funkcí – například správu projektu, doplňování kódu, dokumentace kódu, ladění, napojení na repositář… Já začínal s psaním webů v aplikaci PSPad a NetBeans. Je velmi sympatické, že obě IDE jsou zdarma a stojí za nimi čeští vývojáři.

Je dobré při psaní kódu myslet na jeho udržitelnost, spolehlivost, účinnost a použitelnost. Programátoři často opomíjejí srozumitelnost a čitelnost samotného kódu. Pokud pak čte kód někdo jiný, nebo se k němu sám programátor po delší době vrací, je mnohdy obtížné zpětně zjistit, jak kód pracuje. Nyní si uvedeme několik dobrých praktik vhodný pro obecné programování:

1. Pro názvy používejte stejné a zaběhlé způsoby a používejte je stejně v celém projektu. Například pro názvy konstant se obvykle používají velká písmena abecedy, a pokud je název víceslovný, tak se jako oddělovač používá podtržítko. Příkladem jsou pak konstanty „PI“ nebo „MATH_PI“. Pro proměnné a funkce se pak používají písmena malé abecedy. U víceslovných proměnných se používá tzv. velbloudí notace (anglicky Camel Case) nebo opět podtržítka. Příkladem názvu proměnných

může být „databaze“, „rowCount“, „row_count“. U tříd je to podobné jako s proměnnými, jen se píše i první písmeno velké. Tento způsob zápisu je zaběhlý jako PascalCase. Příkladem je „FileController“. V názvech nepoužívejte diakritiku. Názvy se obvykle píší anglicky.

2. Z názvů proměnných a funkcí by mělo být jasné, k čemu slouží. Například u proměnné „rowCount“ je jasné, k čemu slouží, oproti proměnné „rc“. Vyvarujte se i číslování proměnných. Jen těžko se budete orientovat v kódu s proměnnými „a1“, „a2“ a „a3“. Pro iterace v cyklech jsou běžné názvy proměnných „i“, „j“, „k“. Ve třídách jsou běžné tzv. gettery/settery, které získávají/nastavují obvykle privátní proměnné. U těchto metod se běžně používá prefix get/set. Příkladem jsou pak názvy metod „getActualRowCount“ nebo „setMaxRowCount“. Podobně se pak používají i jiné prefixy. Příkladem je název metody „isValidEmail“, „renderTable“ nebo „drawCircle“ u kterých je také hned jasné, k čemu slouží.
3. Těla funkcí by neměla být dlouhá. Doporučení je maximálně 20–30 řádek kódu. Podobně by neměla ani těla tříd být delší než cca 1000 řádků.
4. Nepoužívejte hluboká zanořování podmínek a cyklů.
5. Udržujte kód jednoduchý. To označuje princip **KISS** (***Keep it simple, Stupid!***)
6. Na začátku si koncepčně rozvrhněte strukturu kódu i adresářovou strukturu projektu.
7. Následujte princip **DRY** (***Don’t Repeat Yourself***) označované i jako **DIE** (***Duplication is Evil***). Jakmile se opakují části zdrojového kódu, není to dobré, opakují se i chyby v nich. Pokud najdete chybu v jednom, musíte ji opravit i v druhém. Podobně je to i s rozšířením funkce a podobně. Kód je méně přehledný a delší.
8. Dokumentujte kód. Zejména je dobré psát dokumentační komentáře pro třídy, metody a proměnné. IDE se vám odmění našeptáváním vaší dokumentace. Pokud se k nějaké třídě vrátíte, lépe poznáte její funkčnost. Z dobře dokumentované třídy lze vytvořit i **API** (***Application Programming Interface***).

1.2.3 Validace

Ve validační fázi kontrolujeme, zdali hotová webová aplikace nebo jen nějaká její iterace neobsahuje chyby a odpovídá zadání. Testovat by se mělo i souběžně s implementací. Testovat se mohou celé aplikace nebo i jen její dílčí části, nebo dokonce i jednotlivé třídy zvlášť (Unit testing). Testovat mohou interní pracovníci, najatí testeři nebo i samotní budoucí uživatelé. V případě, že testování proběhne v pořádku, dochází k předání webové aplikace nebo její patřičné iterace do produkčního prostředí.

1.2.4 Evoluce

Poslední fází je evoluce. Předáním aplikace její „život“ nekončí. Po předání je potřeba řešit mnoho dalších činností, které lze rozdělit do tří základních kategorií:

1. Oprava chyb
2. Tvorba nových funkcí
3. Přizpůsobení se novým technologiím

Při tvorbě webu nesmíme fázi evoluce opomenout. Je potřeba zajistit servis webu i po jeho předání, abychom zajistili jeho dlouhodobý funkční a bezpečný chod a byli jsme připraveni i na případné požadavky zákazníka na rozšíření nebo úpravu funkcí.

1.3 Dobrý web

Znalost jazyka HTML a CSS není záruka pro dobrý web. Dobrý web je takový web, který dobře plní své cíle. Obvykle vytváříme web s cíli, aby jej návštěvníci dobře nalezli, aby se v něm dobře vyznali, aby se vše podstatné řádně dozvěděli a v případě obchodu, aby si produkt či službu koupili. Těmito všeobecnými problematikami se zabývají **SEO** (***Search Engine Optimization***) a přístupnost webových stránek.

1.3.1 Search Engine Optimization

Existuje mnoho způsobů, jak dostat web k jeho návštěvníkovi a SEO je jen jeden z nich. Pro příklad si uvedeme i **SEM** (***Search Engine Marketing***), bannery, adware, emailing, výměny odkazů, publikování v médiích, mobilní marketing, virální marketing, sociální sítě… SEO je způsob, který ovlivňujeme přímo při psaní webové aplikace, a proto se s ním budeme zde zabývat. Jedná se o takovou úpravu zdrojového kódu, můžeme říci optimalizaci kódu, která co nejvíce podpoří nalezení webu například vyhledávačem Google. Tento způsob má své výhody i nevýhody. Oproti SEM může být levnější a dlouhodobější. Koupené kampaně SEM tak mohou působit silněji, ale krátkodobě a jsou obvykle z dlouhodobého hlediska dražší. Také marketingová filosofie je opačná. Při SEO si zákazník hledá inzerenta, kdežto při SEM si inzerent hledá zákazníka.

Jak tedy můžeme zdrojový kód optimalizovat? Nehledejte v tom žádnou vědu, pokud budete při psaní kódu myslet zejména na budoucího návštěvníka webu, máte téměř vyhráno. Nejde jen o správný kód, ale zejména o správný obsah na správných místech. Uveďme si krátký seznam praktik, které mohou crawler boty a poté i vás potěšit. Veškerá zde uvedená doporučení neberme jako povinnosti, patří jen do dobrých praktik a je třeba je u každého webu zvážit. Nezapomínejme, že webová služba je živý organismus a podobně i vyhledávání v něm.

1. Zaregistrujte si web u vyhledávačů. Nastavte si robots.txt a meta robots na každé stránce.
2. Nezapomínejte, že nejdůležitější je kvalitní obsah webu.
3. Mějte stránku rozumně obsáhlou.
4. Pište validní zdrojový kód. Kontrolu si můžete provést na validátoru W3C.
5. Používejte COOL URL.
6. Mějte funkční odkazy.
7. Vytvořte si mapu struktury webu jako sitemap.xml nebo i jako vlastní stránku webu.

Pokud se snažíme dostat se na první stránky vyhledávání (SERP strategie), nezapomínejme, že to ovlivní i vzájemná poloha nalezených slov, umístění nalezených slov (poloha v dokumentu), fráze v URL, umístění v meta značkách pro titulek a popisku. Použití meta pro klíčová slova je zbytečné, například Google i Seznam jej ignorují. Google si umí

sám z obsahu vygenerovat description i titulek. Podobně Google již neřeší velikost desktopové verze webu, ale mobilní verzi ano. Dobrou praktikou je však nenechat uživatele dlouho čekat. Budu se tedy opakovat, ale nejdůležitější je kvalitní obsah.

Na druhou stranu webu nepomůže, když budete mít stránky s malým obsahem, žádnými nebo nefunkčními linky, duplicitními meta značkami, příchozími odkazy nevalných kvalit (link farmy), špatně vyplněným titulkem, s příliš mnoha nadpisy a zvýrazněným písmem, duplikované nebo podobné stránky, neviditelný text s přehršlí klíčových slov atp.

Snažte se tedy tvořit web tak, aby měl kvalitní obsah i ve všech jeho podsekcích a byl populární mezi jeho uživateli a aby se uživatelé chtěli vracet a web doporučovat. Google to ocení.

Při tvorbě webu byste si z marketingového hlediska měli umět odpovědět, jak je na webu řešena odpověď na otázky: Jaké je zaměření firmy? Jaký je hlavní cíl stránek? Jaká je strategie směrování stránek? Jaká je cílová skupina? Kdo je hlavní konkurence? Co nás odlišuje od konkurence?

1.3.2 Přístupnost webových stránek

Touto problematikou se zabývá i přímo W3C a najdete ji tam pod označením **WCAG (Web Content Accessibility Guidelines)**. Jde o sadu mnoha doporučení, která pomohou vytvořit web přístupnější všem. Přístupný web znamená, že se k jeho obsahu řádně dostane co možná nejvíce jeho potenciálních návštěvníků. Uvědomme si, že webové stránky si můžeme prohlížet na mnoha zcela odlišných zařízeních (počítač, notebook, tablet, mobilní telefon, hodinky, televize), které mají různé velikosti a rozlišení a i odlišná vstupní zařízení. Nesmíme zapomenout na diverzitu samotných uživatelů. Někteří lidé mají problém se čtením textů, jiní se ztrácejí v hlubokých strukturách webu, jiní mají zrakové potíže (slepota, barvoslepost), někdo je náchylný k epileptickým záchvatům, někdo má vypnuté obrázky atp. Na to vše je dobré při tvorbě webu myslet. WCAG má všechna svá doporučení rozdělena do tří úrovní shody: A (nízká), AA, AAA (vysoká). Jednoduše řečeno by doporučení A mělo být na webu standardem, doporučení AA by web měl splňovat a doporučení AAA mohl splňovat.

Pro příklad si uvedeme několik zjednodušeně napsaných doporučení:

- netextový obsah musí mít textovou alternativu (úroveň A),
- smysluplné pořadí obsahu textu (úroveň A),
- u navigačních prvků se nespoléhat jen na senzorické vlastnosti jako je tvar, barva, pozice, zvuk… (úroveň A),
- všechny běžné funkce musí být přístupné pomocí klávesnice (úroveň A),
- nepoužívat tři záblesky a další prvky, které mohou způsobovat záchvaty (úroveň A),
- sémantické značky jsou použity v souladu se specifikací (úroveň A),
- u každé stránky musí být definován lidský jazyk (úroveň A),
- pouhým zaměřením nějakého prvku se nesmí měnit kontext webu (úroveň A),

- řádná identifikace chyb, nastanou-li například u špatného vyplnění formuláře (úroveň A),
- správnost syntaxe – validita kódu (úroveň A),
- návrh na zadání bez chyby, systém tipů (úroveň AA),
- konzistentní navigace (úroveň AA),
- vizuální prezentace textu by měla mít kontrastní poměr 4,5:1 (úroveň AA),
- vizuální prezentace textu by měla mít kontrastní poměr 7:1 (úroveň AAA),
- velikost textu je možné měnit bez pomocných technologií (úroveň AAA),
- informace o aktuální stránce ve struktuře webu (úroveň AAA),
- mechanismus pro vysvětlení zkratek či neobvyklých slov (úroveň AAA),
- systém nápovědy (úroveň AAA).

Čtenář byl v těchto kapitolách seznámen se základními principy fungování webu. Měl by kompletně porozumět principům webové komunikace. Cílem těchto kapitol bylo i řádně informovat o možnostech tvorby dobré webové prezentace.

2 Běžné klientské technologie

Klientskými technologiemi chápeme takové, které je schopen webový klient, tedy browser, sám řádně zpracovat. Zabývat se zde budeme jazyky HTML a CSS.

Na celý další výukový materiál by pak vystačil jazyk JavaScript a jeho frameworky. JavaScript je multiplatformní objektově orientovaný jazyk, který je hojně využíván na straně webového klienta, ale i na straně serveru, například v podobě Node.js. Na straně klienta si bez něj webovou stránku již ani neumíme představit, protože právě díky JavaScriptu jsou weby tak interaktivní, jak je nyní známe. Karusely se točí, menu se přizpůsobuje a mění, obrázky z galerie vyskakují a zvětšují se, pozadí se zatmívá, texty připlouvají, produktové dlaždice se vyrovnávají, alarmující okna vyskakují do popředí a mnoho a mnoho dalšího. Zapomenout nesmíme ani na **AJAX (Asynchronous JavaScript and XML)**, díky kterému se například nemusí kvůli změně obsahu načítat celá stránka znovu, nýbrž jen potřebný specifický obsah. Velmi častým takovýmto příkladem jsou našeptávače vyhledávačů.

2.1 HTML

Autorem původního HTML, které vzniklo již počátkem devadesátých let minulého století, je Tim Berners-Lee. O standardy jazyka se stará W3C, na jejichž webu nalezneme i všechny jeho standardy. V současné době je již běžně používaná HTML5. Paralelně, i když původně zamýšlen jako nástupce HTML, je používán i jazyk XHTML, který je přísnější a vychází z pravidel jazyka XML. Pomocí jazyka HTML a jednoduchého neformátovaného textu jsme schopni vytvořit strukturovaný formátovaný multimediální dokument, kterému říkáme hypertext. To vše je za pomocí speciálních značek (tag), které jsou základním stavebním prvkem HTML dokumentu. Značky můžeme do sebe vnořovat a tím vytvářet složité hierarchické struktury. Každá značka má svůj význam a určené použití. Značky můžeme upřesňovat vkládáním atributů. Browser pozná, že se jedná o značku tím, že je obalena ve špičatých závorkách. Značky jsou obvykle párové, tedy mají svůj začátek a konec, který se pozná tak, že před názvem značky je umístěno lomítko. Obsah obalený značkou přejímá vlastnost značky. Tedy například text mezi značkami „b“ bude tučný a bude i patřičně tučně v prohlížeči zobrazen.

Příklad obecného zápisu značky:

```
<název-značky název-atributu=”hodnota-atributu”>Obalený obsah</název-značky>
```

Konkrétní příklady pro nadpis nejvyšší úrovně a následující odstavec, ve kterém je slovo „odstavce“ zvýrazněno tučně:

```
<h1>Nadpis první úrovně</h1>
<p>Toto je text <strong>odstavce</strong></p>
```

Atributy značky se používají zejména u odkazů a obrázků, kde musíme specifikovat cíl odkazu a zdrojový soubor obrázku:

```
<a href="index.html">Home page</a>
<img src="img/zeme.png" alt="Země">
```

Pak existují atributy `id` a `class`, které je možné vložit do každé značky a tím jej jednoznačně rozlišit od ostatních. To se hodí například pro účely JavaScriptu nebo CSS. Význam je víceméně identický, rozdíl mezi nimi je však ten, že hodnota u atributu `id` musí být v rámci dokumentu unikátní. Hodnoty `class` se mohou stejné opakovat bez omezení.

Základní struktura prázdného HTML dokumentu vypadá takto:

```
<!DOCTYPE html>
<html lang="cs">
    <head>
        <!-- Hlavička webu -->
    </head>
    <body>
        <!-- Tělo webu -->
    </body>
</html>
```

Vidíme, že na prvním řádku je určení typu dokumentu vyhovující specifikaci pro HTML 5. Hlavní obalová značka dokumentu je `html`, která má dva potomky `head` a `body`. Informace v hlavičce jsou určené obvykle pro prohlížeče a vyhledávací boty. Najdeme tam například určení kódování, informace o autorovi, popisek, titulek, meta data pro sociální sítě, favicon a další.

Příklad vyplněné hlavičky:

```
<base href="">
<title>HTML Web</title>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<meta name="description" content="Jednoduchá webová stránka, která vznikla za účelem
demonstrace jazyka HTML.">
<meta name="keywords" content="výuka, html">
<meta name="author" content="Michal Bubílek">
<meta name="robots" content="noindex, nofollow">
<meta property="og:title" content="HTML Web" />
<meta property="og:type" content="website" />
<meta property="og:url" content="http://" />
<meta property="og:image" content="img/html.jpg" />
<meta property="og:description" content="Jednoduchá webová stránka, která vznikla za
účelem demonstrace jazyka HTML." />
<meta property="og:site_name" content="Výuka webů" />
<link rel="icon" type="image/png" sizes="32x32" href="img/favicon-32x32.png">
<link rel="icon" type="image/png" sizes="16x16" href="img/favicon-16x16.png">
<link rel="shortcut icon" href="img/favicon.ico">
```

Tělo pak obsahuje vše, co se zobrazuje v hlavním okně prohlížeče. Protože HTML obsah generujeme pomocí PHP, je dobré jej znát. Uvedeme si několik základních značek. Ve výčtu se budou objevovat pojmy jako párová značka, bloková značka atp. Pojďme si je nejprve vysvětlit.

Párová značka je značkou obalovací, která má svůj začátek a konec označený lomítkem. Obsah uvnitř této značky má vlastnost této značky. Příkladem je třeba titulek (<title></title>), odstavec ( ), nadpis (<h2></h2>)… Kdežto značka nepárová nic neobaluje a nemá tedy svou koncovou část. Obvykle slouží jen pro vložení nějaké informace na patřičné místo webu. Příkladem nepárové značky je pak meta (<meta>), link (<link>), obrázek (<img>), odřádkování (<br>)… I nepárové značky můžeme uzavřít: <br />.

Značky dělíme dle typu formátování textu na logické a fyzické. Logické formátování vymezuje význam textu obaleného patřičnou značkou. V textu tak určujeme, co je běžný odstavec, co nadpis, co je důležitější zvýrazněný text, co citace, co zkratka a podobně. U fyzického formátování neurčujeme význam, ale jen vzhled. Můžeme tak určit například tučné písmo, kurzívu, podtržení, horní i dolní index… Například ale u tučného písma neříkáme nic o tom, že je důležitější než jiný text. Browsery mají i pro logické formátování předdefinovaný styl zobrazení, a proto bude nadpis zobrazen výrazněji než běžný text. Celkově si ale vzhled nastavujeme v CSS.

Z hlediska typu zobrazení dělíme značky na blokové a řádkové. To lze v CSS změnit a těchto druhů zobrazení je i více. HTML značky mají ale nastaven nějaký výchozí typ zobrazení. Blokový znamená, že obsah značky se zobrazí jako samostatný obdélník na webové stránce v plné šíři. Řádkové zobrazení chápejme jako text, který může začínat v půlce odstavce a pokračovat přes další tři řádky. Další text pak následuje hned za ním, pokračuje do konce řádku atp.

Nadpisy <h1> až <h6>

- Párové blokové značky s logickým formátováním,
- <h1> je nejdůležitější nadpis, hlavní nadpis, nadpis stránky…,
- <h6> je nejméně důležitý nadpis.

Odstavce

- Párová značka, logické formátování, bloková značka.

Logické formátování textu

- Párové řádkové značky,
- zvýrazněný, důležitý text, obvykle tučný,
- zdůrazněný text, obvykle kurzíva,
- <cite> citace, titulek,
- <code> zdrojový kód,
- <var> proměnná.

Fyzické formátování textu

- Párové řádkové značky,
- <b> tučný text,
- <i> kurzíva,

- `<u>` podtržený text,
- `<sub>` dolní index,
- `<sup>` horní index,
- `<small>` malý text,
- `<big>` velký text,
- `<font>` nastavení fontu.

Odrážkové `<ul>` a číslované `<ol>` seznamy

- Párové blokové značky,
- `<ul>` odrážkový seznam,
- `<ol>` číslovaný seznam,
- `<li>` položka seznamu.

Příklad odrážkového seznamu:

```
<ul>
  <li>První položka odrážkového seznamu</li>
  <li>Druhá položka odrážkového seznamu</li>
</ul>
```

Odkazy `<a>`

- Párová řádková značka,
- atributy `href` (určení cíle odkazu, URL) a `title` (titulek po najetí myši).

Příklad odkazu na externí web a lokální obrázek:

```
<a href="http://www.google.com" title="Google">Klikni na mě</a>
<a href="img/pict.jpg" title="Foto">Zobrazit obrázek</a>
```

Obrázek `<img>`

- Nepárová řádková značka,
- atributy `src` (určení zdroje obrázku) a `alt` (alternativní text).

Příklad:

```
<img src="img/mesto-leto.jpg" alt="Krásný letní den v ulici malého města">
```

Tabulky

- Párové značky,
- `<table>` tabulka,
- `<tr>` řádek tabulky,
- `<th>` buňka záhlaví,
- `<td>` datová buňka.

Příklad jednoduché tabulky:

```
<table>
  <tr>
    <th>Jméno</th>
    <th>Příjmení</th>
  </tr>
  <tr>
    <td>František</td>
    <td>Novák</td>
  </tr>
  <tr>
    <td>Emil</td>
    <td>Nádeníček</td>
  </tr>
</table>
```

Formuláře

- `<form>` Formulář, hlavní obalovací značka, atributy: `action` (URL skriptu, který bude data z formuláře zpracovávat), `method` (budou-li data zaslána metodou `GET` nebo `POST`).
- `<input>` Vstupní políčko, nepárová značka, atributy: `name` (název), `type` (`text`, `password`, `hidden`, `email`…), `value` (výchozí hodnota).
- `<select>` Výběrové menu, párová značka, atribut: name.
- `<option>` Položka výběrového menu, párová značka, musí být přímým potomkem značky `<select>`, atribut: `value` (odesílaná hodnota).
- `<textarea>` Textová oblast, párová značka, atributy: `name`, `rows`, `cols`.
- `<label>` Popisek pole, párová značka, atribut: `for`.
- `<fieldset>` Množina prvků.
- `<legend></legend>` Nadpis.

Příklad formuláře:

```
<form action="form.php" method="POST">
  <fieldset>
    <legend>Login</legend>
    <label for="name">Jméno:</label>
    <input id="name" type="text" name="username" value="">
    <label for="pass">Heslo:</label>
    <input id="pass" type="password" name="password" value="">
    <select id="lang" name="language">
      <option value="cze">Čeština</option>
      <option value="eng">English</option>
      <option value="ger">Deutsh</option>
    </select>
    <input type="submit" name="submit" value="Přihlásit">
  </fieldset>
</form>
```

Kontejnery `<div>` a `<span>`

- Bez grafického efektu,

- bez specifického významu,
- používají se pro tvorbu struktury a layoutu webu,
- `<div>`, párová bloková značka,
- `<span>`, párová řádková značka.

HTML 5 značky

- `<article>` Článek, komentář…,
- `<aside>` například boční text,
- `<canvas>` plátno pro vykreslování her, obrázků…,
- `<cite>` označení titulku díla, ze kterého je citováno,
- `<figcaption>` a `<figure>` popisky k například obrázkům,
- `<header>` hlavička,
- `<footer>` patička,
- `<nav>` hlavní navigace,
- `<section>` obsahová část stránky.

Nejsou zde uvedeny všechny značky. Studenti by si měli případné další dohledat a alespoň vědět, že existují. Nesmíme také zapomenout, že odřádkování ve zdrojovém kódu bude prohlížeč ignorovat a z několika po sobě jdoucích mezer udělá mezeru jednu.

Zdrojový kód:

```
Toto      je              pokusný
text.
```

Výsledek v okně prohlížeče:

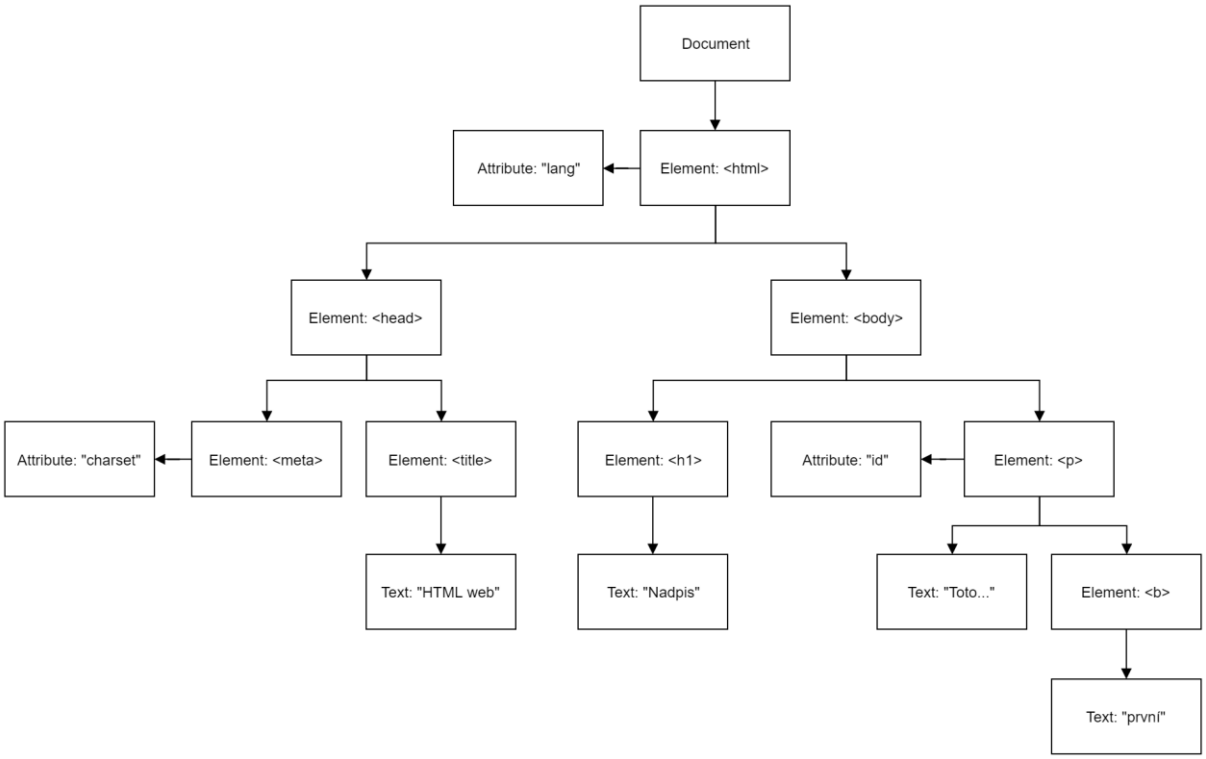
```
Toto je pokusný text.
```

Pokud přeci jen toto potřebujeme, máme možnost díky značkám `<pre></pre>`, `<br>` a HTML entitě ` `, která je popsána níže.

DOM je objektově orientovaná reprezentace HTML dokumentu. Umožňuje přístup k jednotlivým objektům obsahu dokumentu. Můžeme tak upravovat jak obsah, tak strukturu dokumentu. DOM má hierarchickou (stromovou) strukturu, která dokument reprezentuje. Mějme jednoduchou webovou stránku:

```
<!DOCTYPE html>
<html lang="cs">
    <head>
        <title>HTML Web</title>
        <meta charset="utf-8">
    </head>
    <body>
        <h1>Nadpis<h1>
        <p id="první-odstavec">Toto je <b>první</b> odstavec.</p>
    </body>
</html>
```

Pak její DOM bude:



Obrázek 4 DOM ukázkového webu

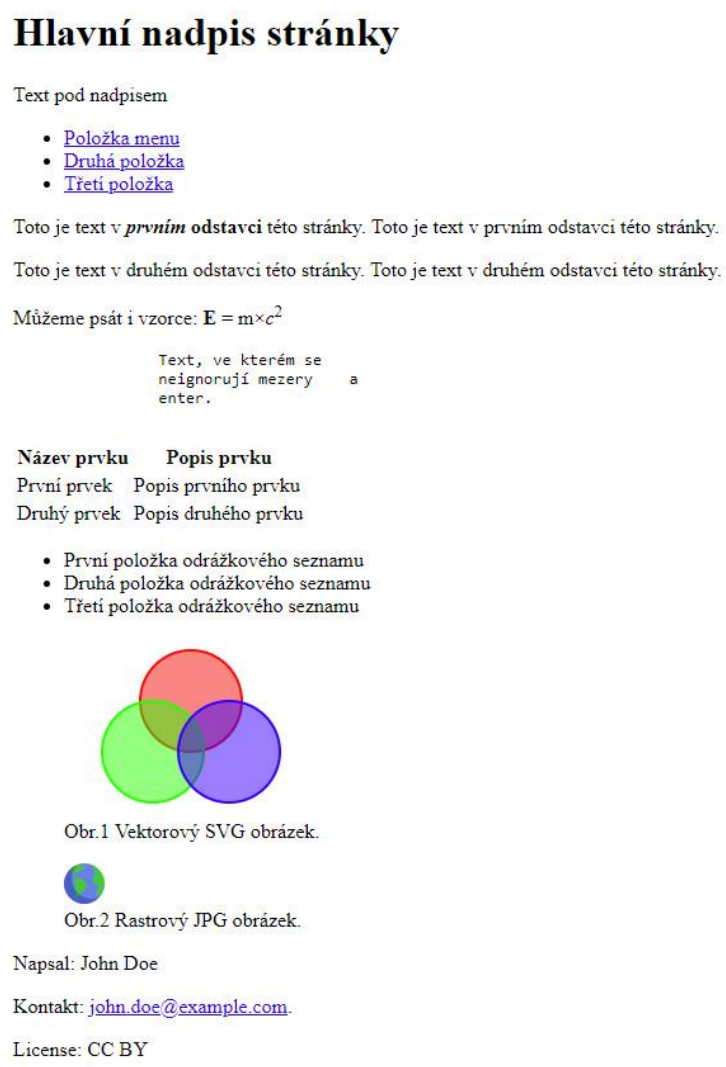
Díky tomuto objektovému modelu můžeme lépe pochopit strukturu dokumentu a různá označení typu potomek elementu, přímý potomek elementu, rodič, element na stejné úrovni, element za elementem a podobně.

Protože jsou některé znaky rezervované pro účely jazyka a některé znaky nejsou běžně na klávesnici, jsou zavedeny HTML entity. Jedná se o speciální řetězce začínající symbolem ampersand (&) a končící středníkem (;). Mezi těmito znaky je název nebo kód entity.

Tabulka 1 Vybrané HTML entity

Výsledek	Název	Entity s názvem	Entity s číslem
&	Ampersand	&amp;	&#38;
	Nedělitelná mezera		
<	Menší než	&lt;	&#60;
>	Větší než	&gt;	&#62;
€	Euro	&euro;	&#8364;
©	Copyright	&copy;	&#169;
∑	Suma	&sum;	&#8721;

Vidíme, že systém značek je jednoduchý a přitom velmi silný nástroj pro vytváření strukturovaného multimediálního obsahu. Pro bližší informace navštivte weby W3C a W3Schools.



Obrázek 5 Ukázkový web psaný čistě v HTML 5 (snímek obrazovky)

2.2 CSS

CSS je jazyk pro definici vzhledu webové stránky. Zpracováván je na straně klienta, kam přichází jeho zdrojový kód. O standardy CSS se stará W3C. V současné době se používá CSS3. CSS můžeme vložit do HTML třemi způsoby, kde každý má své výhody, nevýhody a důvody použití.

Zapsání CSS přímo do HTML značky pomocí atributu `style`:

```
<h1 style="color: red">Nadpis</h1>
```

Zapsání CSS mezi značky `<style>` v hlavičce dokumentu:

```
<style>h1 {color: blue}</style>
```

Vložení externího souboru pomocí odkazu `<link>` umístěného v hlavičce dokumentu:

```
<link href="styl.css" rel="StyleSheet">
```

Platí pravidlo, že pokud je předpis duplicitní, tak platí ten, který je elementu blíže. Pokud by tedy byly použity oba dva první způsoby, tak na webové stránce bude nadpis červený, který přebije tu modrou definici, která by eventuálně přebila definici v externím souboru styl.css.

Třetí způsob je obecně nejvíce doporučovaný. Design se nijak nemíchá do HTML, je tedy oddělen od obsahu a struktury. Výměna externího souboru znamená jiný design bez zásahu do původního html dokumentu. Někdy však, zejména při generování HTML obsahu programem, potřebujeme určovat styl dynamicky, a pak používáme první a druhý způsob. Záleží na způsobu použití. Může to být akceptovatelné. Špatné pro další údržbu by však bylo, kdyby se předpisy CSS například míchaly u dat obsahu webu v databázi.

Psaní předpisů v jazyce CSS je jednoduché. Syntaxe je jednoduchá. Obecný předpis vypadá takto:

```
h1 {
	color: #0000FF;
	font-size: 20px;
}
```

`h1` je selektor, `color` a `font-size` jsou vlastnosti a `#0000FF` a `20px` jsou hodnoty vlastností. Selektorem určujeme, co se bude v dokumentu stylovat a využíváme znalost DOM. Vlastnosti určují, jakou designovou úpravu na vybraném elementu použijeme. Každá vlastnost má svůj obor hodnot, které jí můžeme nastavit. Jednotlivé předpisy od sebe oddělujeme středníky.

2.2.1 Selektory

Představme si, že máme v dokumentu obecně nějaké elementy E a F, které zastupují jakékoliv existující elementy (`h1`, `p`, `table`, `td`, `a`, `img`…), pak pro jejich selekci platí následující základní předpisy.

Tabulka 2 Základní CSS selektory a jejich vysvětlení

Selektor	Určení obsahu v dokumentu
*	jakýkoliv element
E	všechny výskyty elementu E v dokumentu
E, F	všechny výskyty elementu E i F v dokumentu
E F	Element F, který je potomkem elementu E (vertikálně v DOM).
E > F	Element F, který je přímým potomkem elementu E (vertikálně v DOM).
E + F	Element F, který je bezprostředně za elementem E (horizontálně v DOM).

Vybírat si můžeme i dle atributů a jejich hodnot:

Tabulka 3 CSS selektory dle atributů HTML značek

Selektor	Určení obsahu v dokumentu
[atribut]	jakýkoliv element s definovaným atributem
E[atribut="hodnota"]	Element E, jehož atribut je roven určené hodnotě.

E[atribut*="hodnota"]	Element E, jehož atribut obsahuje určený podřetězec.
E[atribut~="hodnota"]	Element E, jehož atribut obsahuje určené slovo.
E[atribut^="hodnota"]	Element E, jehož atribut začíná určeným řetězcem.
E[atribut$="hodnota"]	Element E, jehož atribut končí určeným řetězcem.

Aby bylo z tabulky vše jasné, tak si vysvětlíme, jaký je rozdíl mezi „řetězcem“ a „slovem“. Řetězec obsahuje po sobě jdoucí znaky. Slovo je řetězec, který je buď samostatný bez mezer, nebo je součástí nějakého většího řetězce a je z obou stran obklopen buď mezerou, nebo koncem. Uveďme si například řetězec „CSS je mocný nástroj.“. Tento řetězec obsahuje podřetězce „mo“, „stroj“, „ný ná“ a mnoho dalších. Také obsahuje slova „CSS“, „je“, „mocný“ a „nástroj“.

Tím jsme zdaleka selektory neskončili. Vybírat můžeme přímo i podle atributů `id` a `class`. U značky můžeme napsat i více tříd. Mějme například v dokumentu tento zápis:

```
<div id="odstavce">
    <p class="odstavec lichy">…</p>
    <p class="odstavec sudy">…</p>
    <p class="odstavec lichy">…</p>
</div>
```

Pak můžeme tvořit selektory viz. tabulka níže:

Tabulka 4 CSS selektory pro id a třídy

Selektor	Určení obsahu v dokumentu
#odstavce	Element s id roven hodnotě „odstavec“.
p#odstavec	Tento zápis funguje, ale je zcela zbytečný. Hodnota id může být jen jedna, a tak je zbytečné specifikovat i typ značky.
.lichy	Jakýkoliv element, který má třídu s hodnotou „lichy“.
.odstavec.lichy	Jakýkoliv element, který má třídu s hodnotou „odstavec“ i s hodnotou „lichy“.
p.lichy	Element p, jehož třída má hodnotu „lichy“. Jiné elementy s třídou „lichy“ vybrány nebudou.

Minulý příklad berme jen jako ukázku pro selektory, protože existují i jiná „hezčí“ řešení. Vždy se snažme nepoužívat `id` a `class` zbytečně. V CSS existují i pseudotřídy a pseudoelementy. Pseudotřídy určují speciální stav existujícího elementu. Pseudoelement ovlivňuje jen část existujícího elementu. Pseudoelement může být na základě jeho předpisu do DOM vložen.

Tabulka 5 CSS pseudotřídy

Selektor	Určení obsahu v dokumentu
:link	ještě nenavštívený odkaz
:visited	již navštívený odkaz
:active	Odkaz, na který uživatel klikl, ale ještě nepustil tlačítko myši.
:hover	Element, nad kterým je kurzor myši.

:focus	vybraný odkaz například pomocí klávesnice
:first-child	Jakýkoliv element, který je prvním potomkem svého rodičovského elementu.
:nth-child()	Jakýkoliv element, který je n-tým potomkem svého rodičovského elementu. V závorce udáváme předpis, kolikátý element svého rodiče to je. Například: „1“, „2n+1“…

Předchozí příklad bychom pak mohli zapsat jen čistě a jednoduše takto:

```
<div>
    <p>…</p>
    <p>…</p>
    <p>…</p>
</div>
```

A CSS pro výběr lichých a sudých odstavců, které jsou přímými potomky kontejneru `div` pak takto:

```
div > p:nth-child(2n + 1){
    background: red;
}
div > p:nth-child(2n + 0){
    background: green;
}
```

Tabulka 6 CSS pseudoelementy

Selektor	Určení obsahu v dokumentu
::before	Vloží element před vybraný element.
::after	Vloží element za vybraný element.
::first-line	První řádek vybraného elementu.
::first-letter	První znak vybraného elementu.

Pseudoelementů a zejména pseudotříd existuje více, zde jsem pro ukázku uvedl jen některé, často používané.

CSS nám od verze 3 nabízí i tzv. **Media Query**, které nám přináší podmíněnou selekci obsahu. Podpora prohlížečů je velmi dobrá. Můžeme tak například odstavce stylovat separátně pro obrazovky a tiskárny.

```
@media screen{
  p{background: yellow;}
}
@media print{
  p{background: transparent;}
}
```

Můžeme, a to je velmi často používáno pro tzv. responzivní web, používat i předpisy dle minimální šíře webové stránky:

```
@media only screen and (min-width: 1200px) {
  p{font-size:1.2em;}
}
```

Těchto vlastností, které můžeme podmiňovat, je více. Pro příklad uvedu `max-width`, `orientation`, `max-resolution`…

2.2.2 Vlastnosti a jejich hodnoty

V CSS existuje mnoho vlastností, kterými můžeme ovlivnit vzhled prvků na webové stránce. Opět si vybereme jen některé často používané.

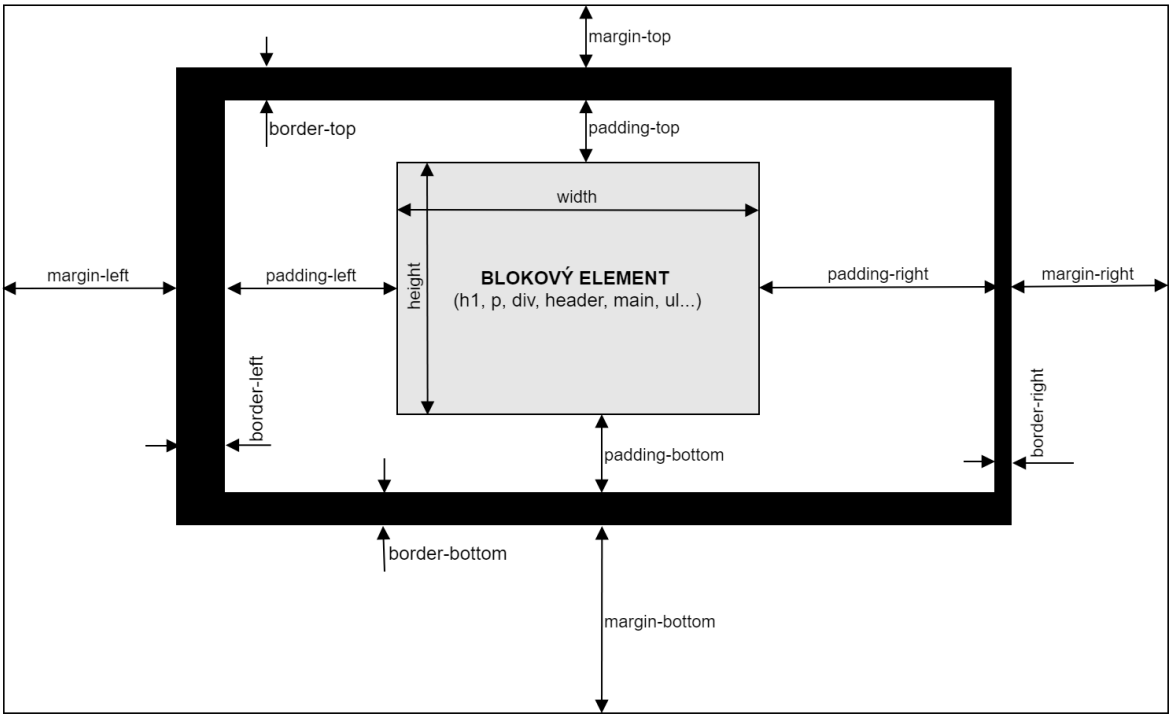
Tabulka 7 Vybrané CSS vlastnosti

Vlastnost	Příklad hodnot	Popis
color	red, #F00, rgb(255,0,0)	barva textu
text-align	left, right, center, justify	zarovnání textu
text-decoration	none, underline, overline, line-through	dekorace textu podtržením, přeškrtnutím…
background	#1122FF url('bg.jpg') no-repeat right top	specifikace pozadí (barva, obrázek, opakování, pozice…) Můžeme určovat i jednotlivé vlastnosti pozadí zvlášť: background-color, background-image, background-repeat, background-position…
font-family	Times New Roman, serif	výběr fontu pro text
font-size	14px, 1.2em	velikost textu
font-weight	normal, bold, 200, 700	tučnost textu
font-style	normal, italic, oblique	styl textu
border	black solid 10px	rámeček (barva, tloušťka, styl) Můžeme určovat i jednotlivé vlastnosti rámečku zvlášť: border-style, border-width, border-color, border-top, border-left…
padding	2px, 2px 4px, 1px 2px 0 0	vnitřní odsazení Můžeme určovat i jednotlivé části odsazení zvlášť: padding-top, padding-right, padding-bottom, padding-left
margin	2px, 2px 4px, 1px 2px 0 0	vnější odsazení Můžeme určovat i jednotlivé části odsazení zvlášť: margin-top, margin-right , margin-bottom, margin-left
display	inline, block, none, inline-block, grid, flex…	styl zobrazení Hodnota „none“ znamená, že element zcela opustí layout.
visibility	visible, hidden, collapse, inherit	viditelnost Hodnota „hidden“ znamená, že element zůstane v layoutu, jen nebude zobrazován.
position	relative, absolute, fixed, static	styl pozicování
top	5px	odsazení elementu shora
right	10%	odsazení elementu zleva

bottom	12mm	odsazení elementu ze spodu
right	2in	odsazení elementu zprava
overflow	auto, hidden, scroll, visible	Způsob nakládání s obsahem, který se nevejde do svého elementu.
z-index	1, 2, 999	Pozice elementu v Z souřadnici 2D zobrazení. Určuje tedy pomyslnou vrstvu, kde menší hodnoty jsou v pozadí a vyšší hodnoty jdou do popředí.
cursor	auto, text, default, pointer, move, help…	styl kurzoru myši

Z tabulky si také můžeme všimnout, že kromě speciálních hodnot se u vlastností zadávají často hodnoty délkové a barvy. Délky obecně dělíme na absolutní (`px`, `mm`, `in`, `pt`…) a relativní (`%`, `em`, `rem`…). Pokud je hodnota `0`, nemusíme psát jednotky. Pokud je hodnota desetinné číslo, píšeme desetinnou tečku. Úvodní nulu psát nemusíme. Barvy se zadávají dvěma způsoby. Jednak existuje 140 slovních vyjádření (`black`, `blue`, `coral`, `darkred`, `transparent`…) a vyjádřit lze také jednoduše pomocí složek RGB, kde z angličtiny R je červená (red), G je zelená (green) a B je modrá (blue). RGB zápisy známe tři: základní např. `rgb(200,100,10)`, s alpha složkou (průhledností) např. `rgba(255,255,255,.5)` a šestnáctkový např. `#FFFFFF` nebo jen zkráceně `#FFF`. Průhlednost se zadává jako poslední čtvrtý parametr v závorce u rgba zápisu. Jedná se o plynulý přechod transparentnosti vyjádřené desetinným číslem z intervalu <0;1>, kde 0 je zcela transparentní a 1 je zcela netransparentní.

Pro lepší pochopení pojmů vnitřní a vnější odsazení si uvedeme schéma, ze kterého je vidět, jak kolem každého blokového elementu definujeme jeho vlastní prostor, do kterého mu nesmí jiný element vstoupit. Výjimky lze samozřejmě obejít u absolutního pozicování.



Obrázek 6 CSS blokový model

CSS také nabízí i CSS entity. Jedná se o kódy pro znaky, které můžeme zobrazit pomocí vlastnosti content. Pro příklad si uvedeme: `&` (0026), `1` (0030), `@` (0040), `A` (0041), `^` (005E), `a` (0061)… Příklad zápisu CSS, kde se bude za každý nadpis vkládat vykřičník:

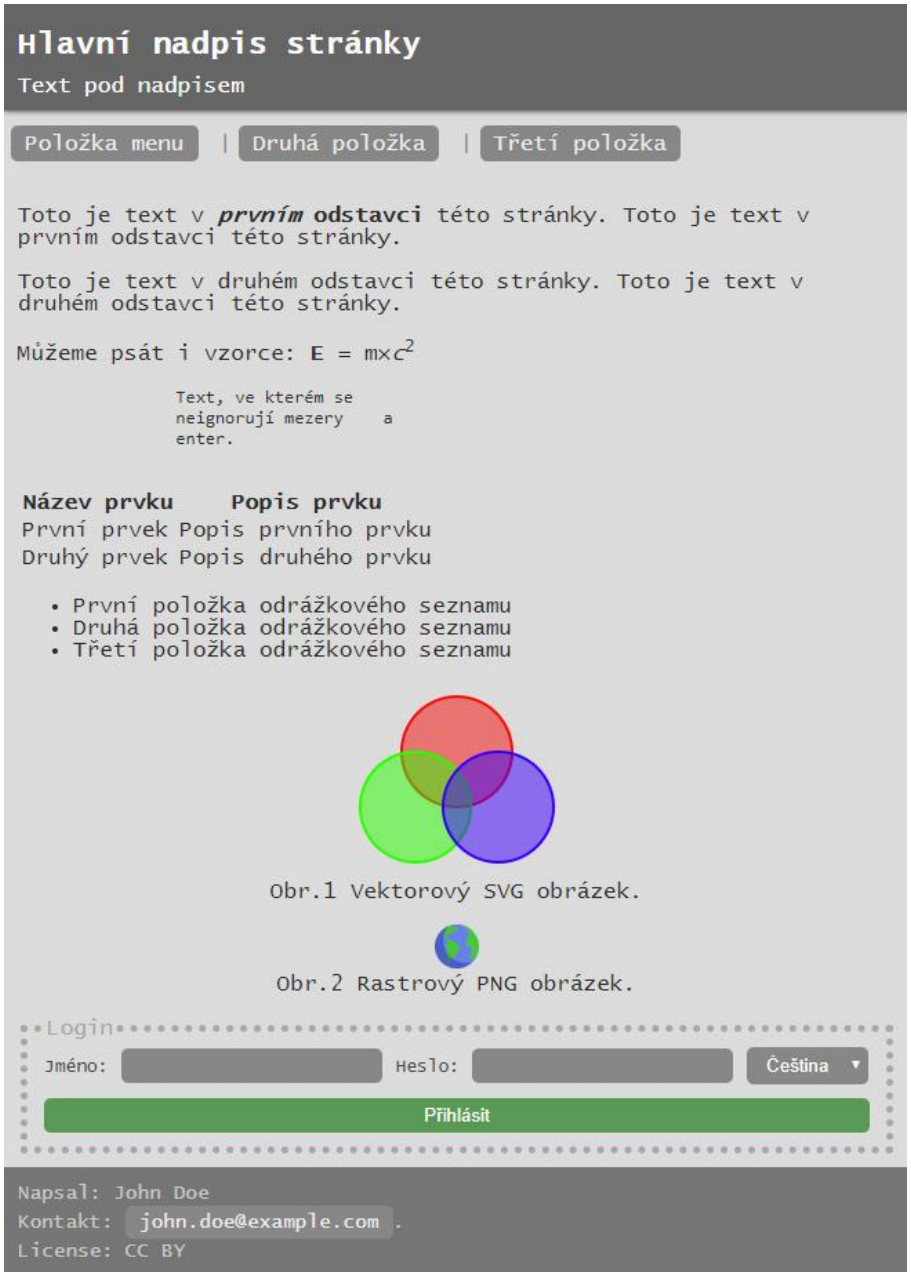
```
h1:after{content: '\0021';}
```

V CSS lze provádět i animace. Můžeme nastavit design klíčovým rámcům a ty poté nechat například ve smyčce. Pro ukázku si ukážeme nekonečně plynule měnící se barvy pozadí stránky mezi barvami červená, modrá a zelená:

```
body{
    animation: menici-se-barva-pozadi 10s linear infinite;
}
@keyframes menici-se-barva-pozadi {
    0%   {background-color: #f00;}
    33%   {background-color: #0f0;}
    66%   {background-color: #00f;}
    100%   {background-color: #f00;}
}
```

Pro další velmi zajímavé efekty je možné využít i CSS vlastnost `filter`, která nám přináší: rozmazání, zostření, barevnost, průhlednost… Také vlastnost `transform` přináší velmi široké možnosti designování s pomocí posouvání, rotace, zmenšování, zvětšování…

CSS je velmi mocný nástroj pro předpis vzhledu nejen pro webové aplikace. Využití CSS najdeme už i u desktopových a mobilních aplikací. Pro zvládnutí dobrého stylování je však potřeba mnoho praxe. Zde jsme si zdaleka neuvedli vše. Spíše se jednalo o malý exkurz do kaskádových stylů. Pro bližší informace navštivte weby W3C a W3Schools.



Obrázek 7 Ukázkový web HTML5 + CSS (snímky obrazovky desktopové a mobilní verze)

3 Běžné serverové technologie

Nyní se dostáváme na stranu serveru. Tam nás, programátory bude nejvíce zajímat jazyk programového interpretu a jazykový prostředek pro systémy řízení báze dat. V našem případě to bude konkrétně PHP a SQL (Structured Query Language). Nebudeme si již vysvětlovat zpracování požadavku, generování a odeslání odpovědi. Budeme se zabývat čistě těmito jazyky a jejich možnostmi. O jazyku SQL se zmíním jen velmi okrajově, protože to je téma velmi komplexní a na vlastní rozsáhlý výukový materiál. Větší váha bude tedy přenesena na PHP, u kterého budeme dále navazovat na objektové programování a sadu komplexnějších příkladů.

3.1 PHP

PHP je skriptovací programovací jazyk určený obvykle ke generování webového obsahu na straně webového serveru. V současné době se na serverech setkáváme běžně s verzemi 5.6, 7.1 a 7.2. Většina příkladů v těchto materiálech budou pracovat na obou těchto verzích. Pokud však vytváříte novou webovou aplikaci, použijte server s PHP verzí 7 a využijte možnosti jazyka PHP této verze. Kromě toho, že PHP 7 je významně rychlejší, nabízí i nové funkce, které se hodí zejména pro objektově orientované programování:

- Deklarace parametrů funkce i se skalárními datovými typy (bool, float, int, string) a jejich typová kontrola,
- určení datového typu návratové hodnoty funkce,
- Null Coalesce Operator ??,
- porovnávací „Spacechip“ operátor <==>,
- pole definované jako konstanty,
- anonymní třídy.
- skupinová use deklarace,
- nové funkce intdiv(),Generator::getReturn(),Closure::call()…,
- datový typ void,
- nastavení viditelnosti (public, private, protected) u třídních konstant,
- spousta fatálních chyb byla konvertována na výjimky,
- všechny E_NOTICE hlášení byly překlasifikovány na E_NOTICE, E_WARNING a E_DEPRICATED.

Opuštěno (Deprecated) v PHP 7 je:

- starý (PHP4) způsob deklarace konstruktoru,
- statické volání nestatických metod,
- salt možnost u funkce password_hash(),
- rozšíření ext/ereg nahrazeno ext/pcre,
- rozšíření ext/mysql nahrazeno ext/mysqli a ext/pdo_mysql.

Více o přepracovaných, nových nebo opuštěných funkcích jazyka a zpětné kompatibilitě naleznete přímo na oficiálním webu php.net.

3.1.1 Základní syntaxe

V příkladech vysvětlujících danou problematiku budu pro názornost pojmenovávat proměnné, funkce a další struktury tak, aby i samotný název čtenáři mnoho napovídal. Čtenář mi jistě promine, že někdy budu používat anglické názvy a jindy zase české. V běžném programování jistě doporučuji vše pojmenovávat dle dohodnutých norem, ve kterých je obvykle koncensus nad angličtinou. Zde budu české názvy používat jen u drobných vysvětlujících příkladů ku prospěchu studenta a pochopení příkladu. Samotnému se mi ježí chlupy na těle při pohledu na diakritiku v názvech proměnných, tříd, rozhraních, konstant… U větších příkladů v kapitole 5 budu pro ukázku „dobrých praktik" používat již jen angličtinu.

PHP se nejčastěji v jednoduchých příkladech vkládá do HTML, a tak tímto začneme i my zde. PHP píšeme mezi značky `<?php` a `?>`. Jiné podoby nedoporučuji. Na serverech mohou být vypnuté a například `<%`, `%>`, `<%=` jsou již ve verzi 7 vyřazené. PHP můžeme vkládat v HTML kamkoliv a kolikráte chceme. Pokud budeme v souboru používat jen PHP, stačí jej na začátku otevřít pomocí `<?php` a ukončovat se již nemusí. Aby webový server předal soubor PHP interpretu a nikoliv rovnou do sítě webovému klientovi, nesmíme zapomenout zapsat souboru příponu php. I když se případně jedná o HTML stránku, tak nesmí mít příponu html nebo htm.

Příklad:

```
<h1><?php echo "Hello world!" ?></h<?php echo "1" ?>>
```

Výsledek:

```
<h1>Hello world!</h1>
```

V příkladu je použit příkaz echo, který se snaží svůj argument (argumenty) převést na řetězec a vypíše je na výstup. Podobně pracuje i příkaz `print` a funkce `printf()` nabízí formátovaný výstup, jak jsme zvyklí například u jazyka C. `Echo` a `print` můžeme zapisovat také jako funkce.

Příklad:

```
echo "Jedna";
echo ("Dva");
print "Tři";
print ("Čtyři");
printf ("Pět");
```

Výstup:

```
JednaDvaTřiČtyřiPět
```

V PHP můžeme používat řádkové (`//`), blokové (`/* … */`) a dokumentační komentáře (`/** … */`). Jakmile začneme psát řádkový komentář, interpret jej chápe automaticky do konce řádku. Blokový komentář musí mít svůj začátek a konec a může jít přes více řádků. Komentáře interpret ignoruje.

Příklad:

```
echo 'Text'; //řádkový komentář
/*
 Blokový komentář
*/
$x = 1 + 2/* + 3 */;
```

V PHP se u všech klíčových slov (echo, if, else, while, break…), tříd a funkcí nerozlišují velká a malá písmena. U názvů proměnných se velikosti rozlišují:

```
$prom = 1;
$Prom = 2;
eCHo $prom; //vypíše 1
```

Příkazy se oddělují středníkem (;).

3.1.2 Proměnné, konstanty, datové typy, přiřazení, operace

Proměnné jsou pojmenovaná místa v paměti, do kterých můžeme ukládat v průběhu běhu programu hodnoty. V PHP se proměnné nemusí deklarovat, nemusí se jim tedy určovat datový typ. Ten si PHP samo dle obsahu k proměnné určí. Proměnné můžeme začít rovnou používat v místě, kde je potřebujeme. Před názvem každé proměnné musí být znak $ (dolar). Název proměnné může obsahovat jen znaky a-b, A-B, 0-9 a _ (podtržítko), přičemž nesmí začínat číslem. U názvu proměnných se rozlišují velikosti písmen. Tomu říkáme, že jsou case-sensitive.

Proměnné vytvořené vně funkcí jsou přístupné jen vně funkcí (globální) a podobně proměnné vytvořené uvnitř funkce jsou přístupné jen uvnitř funkce (lokální). Proměnné uvnitř funkce se po vykonání funkce smažou z paměti. Tomu můžeme předejít pomocí klíčového slova static.

Příklad:

```
$x = 1;
function fce() {
    $y = 2;
    echo "[x=$x,y=$y]";
}
fce();
echo "[x=$x,y=$y]";
```

Výsledek:

```
 [x=,y=2] [x=1,y=]
```

Navíc se dvakrát vypíše upozornění: „Notice: Undefined variable: …“, které nás upozorňuje na pokus získání hodnoty z proměnné, která v paměti není.

Do proměnných ukládáme hodnotu pomocí operace přiřazení (=). Proměnnou, do které ukládáme hodnotu, píšeme vždy vlevo od rovná se. Napravo může být jakýkoliv výraz,

který je nejprve vyhodnocen a jeho hodnota poté uložena. Přiřazení můžeme zřetězovat. Do proměnné můžeme také ukládat referenci na jinou proměnnou pomocí prefixu &.

```
1 + 2 = $x; //špatně
$x = 1 + 2; //dobře x = 1 + 2 = 3
$x = $y = 3; // y = 3 a poté x = 3
$x = ($y = 4) + 5; // y = 4 a poté x = 4 + 5 = 9
$y = 6; // y = 6
$x = &$y; // x = reference na y
echo $x; // 6
$y = 7; // y = 7
echo $x; // 7
```

Hodnoty v proměnných náleží vždy nějakému datovému typu. PHP má skalární datové typy (boolean, integer, float a string), složené datové typy (array, object) a speciální datové typy (resource a null). V PHP 7 jsou tu ještě callable, iterable, void.

Tabulka 8 Vybrané základní datové typy

Datový typ	Příklad hodnot	Popis
boolean	true nebo false	logická hodnota
integer	-1, 0, 1, 2, 3…	celočíselná hodnota
float	3.1415	desetinné číslo
string	"Text"	textový řetězec
array	[1,2,3,4]	pole hodnot

Zdali je proměnná patřičného datového typu, můžeme ověřit pomocí funkcí is_string(), is_bool(), is_float(), is_array(). Pokud chceme otestovat, je-li proměnná nastavená, použijeme isset(). Další užitečnou funkcí je empty(), která zjišťuje, je-li proměnná prázdná. Všechny výše uvedené pomocné funkce vrací logickou hodnotu. Funkce isset() a empty() mohou u někoho vyvolat nejasnosti, proto uvedu příklady. U příkladu je použita funkce var_dump(), která vypíše obsah argumentu i s datovým typem. Pro přehled je v komentáři za příkazem napsán i jeho výsledek.

```
$x = 1;
var_dump(isset($x), empty($x)); //bool(true) bool(false)
var_dump(isset($y), empty($y)); //bool(false) bool(true)
$x = 0;
var_dump(isset($x), empty($x)); //bool(true) bool(true)
$x = null;
var_dump(isset($x), empty($x)); //bool(false) bool(true)
$x = "";
var_dump(isset($x), empty($x)); //bool(true) bool(true)
$x = [1,2,3,4];
var_dump(isset($x), empty($x)); //bool(true) bool(false)
```

```
$x = [];
var_dump(isset($x), empty($x)); //bool(true) bool(true)
```

Shrneme-li to, funkce `isset()` vrací `true`, pokud proměnná existuje a má jinou hodnotu než `null`. V opačných případech vrací `false`. Funkce `empty()` vrací `false`, pokud proměnná existuje a má neprázdnou hodnotu. V opačných případech vrací `true`. Mezi neprázdné hodnoty nepatří: 0 (`int`), 0.0 (`float`), "" (prázdný řetězec), `null`, `false` a prázdné pole.

Pokud chceme proměnnou zrušit, což se nám může hodit, pokud si chceme rychle uvolnit paměť, můžeme použít funkci `unset()`.

Jistě vás i napadne možnost přetypování, tedy potřeby z hodnoty jednoho datového typu vytvořit pokud možno stejnou hodnotu jiného datového typu. Je zajímavé, jak funguje přetypování řetězce na číslo. Uveďme si několik příkladů, které možnosti lépe osvětlí:

```
echo "Převod z boolean:\n";
$bool = true;
var_dump($bool); //bool(true)
var_dump(strval($bool)); //string(1) "1"
var_dump(intval($bool)); //int(1)
var_dump(intval(false)); //int(0)
var_dump(floatval($bool)); //float(1)
echo "\nPřevod z celého čísla:\n";
$int = 1;
var_dump($int); //int(1)
var_dump((string) ($int)); //string(1) "1"
var_dump((boolean) ($int)); //bool(true)
var_dump((float) ($bool)); //float(1)
echo "\nPřevod z floatu:\n";
$float = 1.2;
var_dump($float); //float(1.2)
var_dump((string) ($float)); //string(3) "1.2"
var_dump((boolean) ($float)); //bool(true)
var_dump((int) ($float)); //int(1)
var_dump((int) ceil($float)); //int(2)
echo "\nPřevod z řetězce:\n";
$text = "5 kg";
var_dump($text); //string(4) "5 kg"
var_dump(boolval($text)); //bool(true)
var_dump(intval($text)); //int(5)
var_dump(floatval($text)); //float(5)
var_dump(explode(" ", $text));
/*
array(2) {
  [0]=>
  string(1) "5"
  [1]=>
  string(2) "kg"
}
*/
echo "\nPřevod z pole:\n";
var_dump($pole);
/*
```

```
array(4) {
  [0]=>
  int(2)
  [1]=>
  int(4)
  [2]=>
  int(6)
  [3]=>
  int(8)
}
*/
$pole = [2, 4, 6, 8];
var_dump(boolval($pole)); //bool(true)
var_dump(intval($pole)); //int(1)
var_dump(floatval($pole)); //float(1)
var_dump(implode(",", $pole)); //string(7) "2,4,6,8"
```

Možnosti převodu jsou velmi široké. Můžeme použít funkce `intval()`, `strval()`..., nebo před výraz, který chceme přetypovat, vložit název výsledného datového typu v kulatých závorkách. Pokud převádíme celočíselné číslo na desetinné, nemusíme řešit nic speciálního. Opačně již musíme počítat se ztrátou desetinné části. Pomoci si však v případě potřeby můžeme zaokrouhlovacími funkcemi `round()`, `floor()` a `ceil()`. Velmi pěkný je převod z řetězce na číslo, kdy se PHP snaží rozpoznat číslo z počátku řetězce. Pro převod z pole na řetězec a z řetězce na pole můžeme použít funkce `implode()` a `explode()`.

3.1.2.1 Logické hodnoty

Logická hodnota může nabývat pouze stav `true` (pravda) nebo `false` (nepravda). Krom funkcí, které vrací logickou hodnotu, máme i logické operace, jejichž výsledkem je také `true` nebo `false`.

Tabulka 9 Porovnávací operace

Operace	Význam
==	true, když jsou hodnoty shodné
===	true, když jsou hodnoty identické, tedy shodné v hodnotě i datovém typu
!=	true, když nejsou hodnoty shodné
<>	true, když nejsou hodnoty shodné
!==	true, když nejsou hodnoty shodné, nebo se nerovnají datové typy
>	true, když je levá hodnota větší než pravá
<	true, když je levá hodnota menší než pravá
>=	true, když je levá hodnota větší nebo rovná pravé
<=	true, když je levá hodnota menší nebo rovná pravé

Logické výrazy mohou být komplikovanější než jen jedno porovnání hodnot. Pokud potřebujeme složit větší výraz z několika základních, můžeme použít logické spojky a priority určovat pomocí kulatých závorek.

Tabulka 10 Logické operace

Spojka	Význam
!	unární negace
and	Logický součin. true, když jsou oba výrazy true.
or	Logický součet. true, když je alespoň jeden výraz true.
xor	Exklusivní součet. true, když jsou výrazy různé.
&&	Logický součin. true, když jsou oba výrazy true.
||	Logický součet. true, když je alespoň jeden výraz true.

Rozdíl mezi `and`/`or` a `&&`/`||` je v prioritách. Seznam operací dle priorit od nejvyšší:

1. !
2. &&
3. ||
4. =
5. and
6. xor
7. or

Praktický příklad:

```
var_dump(true || true || false); // true
var_dump(true && false && false); // false
var_dump(false || true && false); // false
var_dump(false || true && true); // true
var_dump(true and false or false); // false
var_dump(true xor false); // true
var_dump(true xor true); // false
$x = true and false; // (x = true) and false
var_dump($x); // true
$x = true && false; // x = (true && false)
var_dump($x); // false
```

Také je dobré neopomenout, že jakmile je výsledná hodnota výrazu známa, zbytek výrazu je již nevyhodnocuje. Z toho by si programátor měl vyvodit dvě pravidla:

1. Komplikovanější operace u logických výrazů umisťovat na konec.
2. Vyvarovat se přiřazovacím operacím ve výrazech, u kterých není jasné, že se provedou.

Příklad:

```
$x = 1;
var_dump(true || ++$x == 2); // true
var_dump($x); // 1
var_dump(false || ++$x == 2); // true
var_dump($x); // 2
var_dump(false || $x++ == 2); // true
var_dump($x); // 3
```

3.1.2.2 Čísla

V PHP můžeme zapsat čísla ve dvojkové, osmičkové, desítkové a šestnáctkové soustavě. Každý zápis má svá pravidla.

Tabulka 11 Struktura zápisu celočíselných hodnot pro různé číselné soustavy

Soustava	Struktura čísla
dvojková	0[bB][01]+
osmičková	0[0-7]+
desítková	[1-9][0-9]* | 0
šestnáctková	0[xX][0-9a-fA-F]+

Celočíselná (integer) čísla jsou vždy znaménková z intervalu určeného předdefinovanými PHP konstantami <PHP_INT_MIN;PHP_INT_MAX >. Konstanty se mohou lišit na 32b a 64b systémech.

Desetinná čísla se zapisují s desetinnou tečku (2.3, -5.1, .564) a může se použít i zápis s uvedením exponentu (1E2, -1.4E5, 123E-4).

V PHP máme 6 aritmetických operátorů.

Tabulka 12 Aritmetické operátory

Operace	Význam
+	unární plus (kladná hodnota) nebo sčítání
-	unární mínus (záporná hodnota) nebo odčítání
*****	násobení
/	dělení
%	zbytek po dělení
**	exponent

Existuje více variací na tyto operace. Známe i inkrementaci (++) a dekrementaci (--), u kterých rozlišujeme variantu pre a post. Pak máme zkrácené zápisy (+=, -=, /=, *=). Pro vlastní určení priorit můžeme použít standardní kulaté závorky. Nejlépe si to ukážeme na příkladech.

```
$x = 2;
$y = 3;
$z = $x + $y; //z = 2 + 3 = 5
$z = $x + $y * 2; // z = 2 + (3 * 2) = 8
$z = ($x + $y) * 2; // z = (2 + 3) * 2 = 10
$x++; // x = x + 1 = 2 + 1 = 3
$z = ++$x - $y; // nejprve x = x + 1 = 3 + 1 = 4 a poté z = 4 - 3 = 1
$x += $y; // x = x + y = 4 + 3 = 7
$x /= 2; // x = x / 2 = 3.5
$x = $z % 2; // x = 1 (zbytek po dělení 7/2)
$z = 3 + -$x++; // z = 3 + (-1) = 2 a poté x = x + 1 = 1 + 1 = 2
$z = $x**$y; // z = x * x * x = 2 * 2 * 2 = 8
$z -= 2; // z = z - 2 = 8 - 2 = 6
```

```
$z = -$x + -(--$y); // nejprve y = y – 1 = 3 – 1 = 2 a poté z = -2 + -2 = -4
```

Pro celočíselné hodnoty máme k dispozici bitové operace.

Tabulka 13 Bitové operace

Operace	Význam
&	AND - bitový součin
|	OR - bitový součet
^	XOR – bitový exklusivní součet
~	NOT – bitová negace
<<	bitový posun vlevo – násobení dvěma
>>	bitový posun vpravo – dělení dvěma

3.1.2.3 Řetězce

Práce s řetězci je velmi jednoduchá a intuitivní. Textový řetězec musí být uvozen a ukončen buď uvozovkami ("), nebo apostrofy ('). V textovém řetězci obaleném uvozovkami můžeme používat jednoduše apostrofy, a pokud chceme uvozovky, musíme použít tzv. escape sekvenci. To platí i obráceně. V uvozovkách můžeme použít escape sekvence \n, \r, \t, \v, \\, \" a další. V řetězcích obalených apostrofy můžeme použít jen \\ a \'. Další rozdíl je v tom, že proměnné v textovém řetězci obaleném uvozovkami jsou nahrazovány svým obsahem, kdežto u řetězců v apostrofech nikoliv. Textové řetězce se spojují operátorem tečka (.). Tak jako u zkrácených aritmetických operací, můžeme použít i zkrácený zápis pro spojení řetězců (.=), který značí přilepení řetězce na konec původního řetězce. K textovým řetězcům můžeme přistupovat jako k poli znaků. To vše si pro názornost uvedeme v příkladu:

```
echo 'Zde píši "uvozovky".\n';
echo "Zde píši 'apostrofy'.\n";
echo 'Zde píši "uvozovky" i \'apostrofy\'.' . "\n";
echo "Zde píši \"uvozovky\" i 'apostrofy'.\n";
echo "Spojujeme" . " " . "více " . 'řetězců' . ' do jednoho.' . "\n";
$sloveso = 'je';
echo "Toto $sloveso řetězec.\n";
echo "Toto $sloveso[0]e řetězec.\n";
echo 'Toto $sloveso řetězec.' . "\n";
echo 'Toto ' . $sloveso . ' řetězec.' . "\n";
```

Výstup:

```
Zde píši "uvozovky".\nZde píši 'apostrofy'.
Zde píši "uvozovky" i 'apostrofy'.
Zde píši "uvozovky" i 'apostrofy'.
Spojujeme více řetězců do jednoho.
Toto je řetězec.
Toto je řetězec.
Toto $sloveso řetězec.
Toto je řetězec.
```

PHP disponuje mnoha funkcemi pro práci s řetězci. Můžeme tak v řetězci vyhledávat, získávat podřetězce, převádět na pole a mnoho dalších funkcí.

Tabulka 14 Vybrané funkce pro práci textovými řetězci

Funkce	Popis
explode()	Převádí řetězec na pole prvků. Můžeme si určit oddělovací řetězec.
html_entity_decode()	Převádí HTML entity v řetězci na jejich korespondující znaky.
htmlentities()	Znaky v řetězci, které je možné konvertovat, konvertuje na jejich HTML entity.
htmlspecialchars()	Všechny speciální znaky (ampersand, uvozovky, apostrofy, menší než, větší než) konvertuje na HTML entity.
htmlspecialchars_decode()	inverzní funkce k htmlspecialchars()
str_replace()	Nahrazuje podřetězce v řetězci jinými řetězci.
strip_tags()	odstranění HTML a PHP značek z řetězce
strlen()	Vrací délku řetězce.
strpos()	Vrací pozici prvního nalezení hledaného podřetězce.
strtolower()	Převede všechny znaky A-Z na a-z.
substr()	Vrátí podřetězec řetězce.
trim()	Odstraní neviditelné znaky (mezery, nové řádky, tabulátor…) ze začátku i konce řetězce.

3.1.2.4 Pole

PHP má velmi flexibilní pole. Pro připomenutí uvedu, že pole je množina prvků, kde každý má svou hodnotu a klíč. Pomocí klíče prvku se v poli dostaneme k jeho hodnotě. U pole v PHP nemusíme udávat počet prvků. Pole mohou mít smíšené datové typy hodnot svých prvků. Pole mohou být asociativní a klíče prvků mohou být také smíšené. Pole mohou být vícerozměrná.

Tabulka 15 Operace s poli

Operace	Význam
+	sjednocení polí (UNION)
==	true, když levé i pravé pole mají stejné prvky (klíče i hodnoty)
===	true, když levé i pravé pole mají stejné prvky i ve stejném pořadí
!=	true, když pole nejsou shodná
<>	true, když pole nejsou shodná
!==	true, když pole nejsou identická (prvky ani pořadí)

Analyzujte rozdíl mezi operací `+` a `array_merge()`.

PHP má mnoho desítek funkcí, které s poli pracují. Některé velmi používané funkce si uvedeme.

Tabulka 16 Vybrané funkce pro práci s poli

Funkce	Popis

array()	Vytvoří pole.
range()	Vytvoří pole prvků v daném rozmezí hodnot.
count()	Vrací počet prvků pole.
in_array()	Zjistí, je-li prvek s hledanou hodnotou v poli.
array_key_exists()	Zjistí, je-li prvek s hledaným klíčem v poli.
sort()	Seřadí prvky pole. (Řadících funkcí existuje mnoho, řadit můžeme vzestupně, sestupně, podle hodnot, podle klíčů, se zachováním vazby klíč – hodnota…)
array_merge()	Sloučí více polí do jednoho.

Každé pole má svůj vnitřní ukazatel na aktuální prvek. Pomocí funkce `current()` získáme hodnotu aktuálního prvku. Pomocí funkcí `end()`, `prev()`, `next()` a `reset()` se můžeme v poli po prvcích pohybovat.

Pokud neuvedeme při vytváření prvků pole jejich klíče, PHP nám automaticky vytvoří klíče číselné. Nezapomínejme, že podobně jako u jiných programovacích jazyků má první prvek klíč s hodnotou 0 (nula). Na jednotlivé prvky pole se dostáváme pomocí hranatých závorek (`[` a `]`). Pokud při ukládání prvku do pole neuvedeme v hranaté závorce hodnotu klíče, bude prvek přidán na konec pole a klíč se automaticky vygeneruje jako další číselný. Pokud hodnotu klíče uvedeme a v poli se prvek s uvedeným klíčem nachází, tak se jeho hodnota přepíše na novou. Pokud takový klíč v poli není, vytvoří se nový prvek pole se zadaným klíčem a hodnotou. Více to osvětlí příklad na jednorozměrná pole:

```
var_dump(array(1, 2, 3));
var_dump([1, 2, 3]);
var_dump([1 => 1, 2, 3]);
var_dump(['red' => 'červená', 'green' => 'zelená', 'blue' => 'modrá']);
$pole = [1, 2.2, 'A'];
var_dump($pole[2]);
$pole[2] = 'B';
var_dump($pole[2]);
$pole[] = 'C';
var_dump($pole[3]);
$pole[55] = 'D';
var_dump($pole[55]);
var_dump($pole);
```

Výsledek:

```
array(3) {
  [0]=>
  int(1)
  [1]=>
  int(2)
  [2]=>
  int(3)
}
array(3) {
  [0]=>
  int(1)
```

```
  [1]=>
  int(2)
  [2]=>
  int(3)
}
array(3) {
  [1]=>
  int(1)
  [2]=>
  int(2)
  [3]=>
  int(3)
}
array(3) {
  ["red"]=>
  string(9) "červená"
  ["green"]=>
  string(7) "zelená"
  ["blue"]=>
  string(6) "modrá"
}
string(1) "A"
string(1) "B"
string(1) "C"
string(1) "D"
array(5) {
  [0]=>
  int(1)
  [1]=>
  float(2.2)
  [2]=>
  string(1) "B"
  [3]=>
  string(1) "C"
  [55]=>
  string(1) "D"
}
```

Vícerozměrná pole jsou taková pole, jejichž prvky jsou opět pole a případně jejich prvky zase pole atp. Pro jednoduchost si dvourozměrné pole představme jako obdélníkové hrací pole, kde má každé políčko svou souřadnici zadanou svou horizontální a vertikální pozicí. Více jak třírozměrná pole si lidé špatně představují a není k tomu vlastně ani důvod si tato pole představovat. Je potřeba si jen prakticky uvědomit, že abychom se dostali k hodnotě daného prvku, potřebujeme znát více klíčů. Například jako kdybychom měli velkou databázi všech existujících automobilů. Abychom se dostali k tomu konkrétnímu, musíme znát jeho výrobce, model, karoserii, motorizaci… Uveďme si příklad pro dvourozměrné pole.

```
$pole = [['A','B','C'],['D','E','F']];
var_dump($pole);
echo $pole[0][2]; //C
```

Výsledek pak vypadá takto:

```
array(2) {
  [0]=>
  array(3) {
    [0]=>
    string(1) "A"
    [1]=>
    string(1) "B"
    [2]=>
    string(1) "C"
  }
  [1]=>
  array(3) {
    [0]=>
    string(1) "D"
    [1]=>
    string(1) "E"
    [2]=>
    string(1) "F"
  }
}
C
```

PHP disponuje i několika vestavěnými superglobálními poli, tedy poli, která jsou přístupná odkudkoliv ve vašem zdrojovém kódu. U většiny z nich samotné PHP připraví před vykonáním vašeho skriptu jejich prvky, které můžete ve skriptu používat. Takováto pole poznáme jednoduše, jsou připravená jako proměnné, které začínají obvykle podtržítkem (_).

Tabulka 17 Superglobální pole

Pole	Popis
$GLOBALS	Pole globálních proměnných, ke kterému můžeme přistupovat i v těle funkcí. Klíčem prvku je název proměnné, hodnotou je pak její hodnota.
$_SERVER	Pole informací ze serveru. Najdeme v něm IP adresu serveru, název software webového serveru, protokol, URI, ale i adresu, ze které se na server přišlo a další data…
$_REQUEST	data odeslaná formulářem
$_GET	data odeslaná formulářem metodou GET
$_POST	data odeslaná formulářem metodou POST
$_FILES	informace o souborech odeslaných formulářem na server
$_COOKIE	pole s uloženými COOKIE
$_SESSION	Pole pro data, která jsou perzistentní v rámci sezení.

Není bezpečné číst data odeslaná formulářem přímo z polí `$_GET`, `$_POST` a `$_REQUEST`. Data mohou být podvržena. K těmto polím přistupujte pomocí funkce `filter_input()`, kde můžeme vstupní data jak validovat (ověřit jejich podobu), tak sanitizovat (upravit

do požadované podoby). Tímto způsobem si můžeme zajistit, že získáme opravdu základní řetězec, číslo, email…

```
$email = filter_input(INPUT_GET, 'email', FILTER_SANITIZE_EMAIL);
```

Server ke každé vygenerované stránce přistupuje individuálně, tedy veškeré proměnné jsou po skončení skriptu smazány z paměti a server je připraven na obsluhu dalšího požadavku. Mnohdy však potřebujeme hodnoty v proměnných zachovat do další stránky. Příkladem je udržení přihlášení uživatele, vložené produkty v košíku a podobně. K tomuto účelu slouží pole `$_SESSION`. Na začátku skriptu, ještě před jakýmkoliv výpisem, tedy než se pošle klientovi HTTP hlavička, se musí session nastartovat funkcí `session_start()`. Poté se již jen zapisuje a čte z perzistentního pole `$_SESSION`. Pustíme nejprve a.php:

```
<?php
session_start();
$_SESSION['x'] = 'X';
```

Poté pustíme skript b.php:

```
<?php
echo $_SESSION['x'];
```

Hodnota „X“ se v poli pod klíčem „x“ uchová i pro b.php, kde je vypsána.

3.1.2.5 Konstanty

Konstanty oproti proměnným si jen jednou nadefinujeme a poté nesmíme jejich hodnoty měnit. Konstanty můžeme definovat pomocí funkce `define()` nebo klíčového slova `const`. Konstanty definované mimo funkce můžeme ve funkci použít a i obráceně – konstanty definované uvnitř funkce můžeme použít vně. Toto nelze u definice pomocí `const`. Konstantu voláme buď jednoduše svým názvem, nebo funkce `constant()`. Názorně je to ukázáno v příkladu:

```
define("KONSTANTA_1", "1\n");
const KONSTANTA_2 = "2\n";
echo constant("KONSTANTA_1"); //1
echo KONSTANTA_2; //2
fce();
echo KONSTANTA_1; //1
echo KONSTANTA_2; //2
echo KONSTANTA_3; //3
const KONSTANTA_4 = ["A\n", "B\n", "C\n"];
echo KONSTANTA_4[1]; //B
function fce() {
    define("KONSTANTA_3", "3\n");
    //const KONSTANTA_4 = "4\n"; //nelze
    echo KONSTANTA_1; //1
    echo KONSTANTA_2; //2
    echo KONSTANTA_3; //3
}
```

3.1.3 Řídící struktury

Do řídících struktur patří podmínky a cykly. Podmínky se nám umožňují program větvit, cykly umožňují části kódu opakovat. Základní podmínka `if`, která je používána většinou programovacích jazyků, platí i v PHP. Tuto podmínku lze zřetězovat, vnořovat, definovat i alternativní větev… Argument v podmínce je vždy vyhodnocen jako `boolean`. Ukažme si několik příkladů:

```
$x = -3;
if (is_numeric($x)) {
    if ($x == 0) {
        echo 'nula';
    } elseif ($x > 0) {
        echo 'kladné';
    } else {
        echo 'záporné';
    }
} else {
    echo 'není číslo';
}
```

Někdy je efektnější při více větvích použít `switch`. U něho můžeme použít zřetězení operací (kaskády), dokud z větve nevyskočíme příkazem `break`.

Mějme kód:

```
switch ($x) {
    case 'a':
    case 'b':
    case 'c':
        echo 'X';
        break;
    case 'd':
        echo 'Y';
    case 'e':
        echo 'Z';
        break;
    default:
        echo '-';
}
```

Pak jeho výstup v závislosti na proměnné `x` bude takovýto:

Tabulka 18 Výstup struktury switch z uvedeného příkladu

Proměnná x	Výstup
a	X
b	X
c	X
d	YZ
e	Z
cokoli jiného	-

Poslední podmínkou je ternární operátor, který můžeme chápat jako inline podmínku. Tedy v místě volání se přímo objeví dle vyhodnocení logického výrazu jedna ze dvou hodnot. Výhodné je toto použít právě na místech, kde se očekávají hodnoty, které určujeme podmínkou a nechceme program rozepisovat na několik řádků. Syntaxe ternárního operátoru je jednoduchá:

```
logický_výraz ? hodnota_při_true : hodnota_při_false;
```

Ternární operátory je možné do sebe vnořovat. Pojďme si to ukázat na příkladu, kde tutéž situaci napíšeme jak standardní podmínkou, tak pomocí ternárního operátoru:

```
$c = 1;
echo "Máme $c ";
if ($c == 1) {
    echo "úl";
} else if ($c > 1 && $c < 5) {
    echo "úly";
} else {
    echo "úlů";
}
echo ". ";

echo "Máme $c " . ($c == 1 ? "úl" : ($c > 1 && $c < 5 ? "úly" : "úlů")) . ".";
```

Ternární operátor používejme s rozmyslem. Program by měl být stále dobře čitelný a pochopitelný. To řeší strategie KISS.

3.1.4 Funkce

V PHP jsou stovky již připravených funkcí, které můžeme obratem používat. Není třeba vkládat do úvodů zdrojových kódů nějaké knihovny a podobně. Automaticky máme přístup k funkcím pro práci s poli, textovými řetězci, datem a časem, čísly, souborovým systémem a mnoho dalších okruhů činností. Jednotlivé funkce jsou dobře popsány v manuálu PHP. Není je tedy třeba vysvětlovat.

Funkce si můžeme vytvářet vlastní. Zejména v OOP je velmi zásadní znalost práce s funkcemi. Funkce se deklaruje pomocí klíčového slova `function`, za kterým následuje název funkce, v kulatých závorkách parametry a za nimi ve složených závorkách samotné tělo funkce. V PHP se neudává návratový typ. Pokud tvoříme funkci bez parametru, stejně kulaté závorky uvádíme. Funkci voláme jejím názvem, za kterým vždy uvádíme i kulaté závorky. Příklad jednoduché funkce bez parametrů, která po svém zavolání vypíše aktuální čas:

```
printTime();

function printTime(){
    echo date("H:i:s") . "<br>";
}
```

Všimněme si, že funkci můžeme deklarovat až po jejím volání. Funkce nemusí nic vracet. Další příklad ukáže funkci s proměnným počtem parametrů, která bude vracet jejich součet:

```
echo "Výsledek je " . suma(1) . ".<br>";
echo "Výsledek je " . suma(1, 2) . ".<br>";
echo "Výsledek je " . suma(1, 2, 3) . ".<br>";

function suma($x, $y = 0, $z = 0) {
    return $x + $y + $z;
}
```

Kde výsledek je:

```
Výsledek je 1.
Výsledek je 3.
Výsledek je 6.
```

Funkce má vlastní paměťový prostor. Proměnné vytvořené ve funkci nejsou přístupné vně a obráceně.

```
$x = 1;
echo $x;
fce();
echo $x;

function fce() {
    $x = 2;
    echo $x;
}
```

Výstup:

```
121
```

Můžeme však použít & (ampersand) pro adresu proměnné (ukazatel):

```
$x = 1;
echo $x;
fce($x);
echo $x;

function fce(&$x) {
    $x = 2;
    echo $x;
}
```

Výstup:

```
122
```

Příkaz return ukončí provádění funkce a vyhodnocený výraz zapsaný hned za returnem se vloží do místa volání funkce. Pokud tedy existuje nějaký kód za returnem, nikdy se

nevykoná. Efektně jej však můžeme využít v podmínkách. Příkazů return může být ve funkci více. Zase však pozor na přehlednost kódu. Oproti jiným jazykům může jedna funkce v PHP vracet hodnoty různých datových typů. Výše uvedené vlastnosti si předveďme v příkladu:

```
if (($x = divide(1, 2)) !== false) {
    echo $x;
}

function divide($x, $y) {
    if ($y == 0) {
        return false;
    }
    return $x / $y;
}
```

Výsledek:

```
0.5
```

Toto je trochu krkolomný příklad, ale jednoduše demonstruje vše potřebné. Zrovna operace dělení nulou by se dala řešit vyhozením výjimky. O tomto ale až později.

PHP podporuje anonymní funkce, tedy funkce, u kterých nezadáváme jejich název. Jednoduchý příklad:

```
$fce = function($n) {
    print($n * $n);
};
$fce(2);
```

Výstup:

```
4
```

V tomto příkladu se naše jednoduchá anonymní funkce uložila do proměnné, kterou je poté hned volána. Pojďme si ukázat něco praktičtějšího, tedy použití anonymní funkce v parametru jiné funkce. Pomocí funkce `array_map()` a naší anonymní funkce si vypíšeme jednotlivé prvky zadaného pole:

```
array_map(function($item) {
    printf("%s ", $item);
}, array(1, 2, 3, 4, 5));
```

Výstup:

```
1 2 3 4 5
```

3.1.5 Chyby a ladění kódu

Ladění PHP kódu se může zdát obtížné, je-li spouštěn na straně serveru. Samotné PHP nám nenabízí žádný sofistikovaný způsob ladění. Použít můžeme hotová externí řešení

jako například Tracy z Nette, která je také známá pod označením „Laděnka“. Samotné PHP nám nabízí funkce var_dump(), var_export() a debug_zval_dump():

```
$promenna = 1.1;
var_dump($promenna);
debug_zval_dump($promenna);
var_export($promenna);
echo "\n";
$pole = array(1, 1.1, '2.1', true);
var_dump($pole);
debug_zval_dump($pole);
var_export($pole);
```

Výsledek:

```
float(1.1)
double(1.1) refcount(2)
1.10000000000000008881784197001252323389053344726562 5
array(4) {
  [0]=>
  int(1)
  [1]=>
  float(1.1)
  [2]=>
  string(3) "2.1"
  [3]=>
  bool(true)
}
array(4) refcount(2){
  [0]=>
  long(1) refcount(1)
  [1]=>
  double(1.1) refcount(1)
  [2]=>
  string(3) "2.1" refcount(1)
  [3]=>
  bool(true) refcount(1)
}
array (
  0 => 1,
  1 => 1.10000000000000008881784197001252323389053344726562 5,
  2 => '2.1',
  3 => true,
)
```

Funkce var_dump() nám zobrazí hodnotu a datový typ svého argumentu. Funkce debug_zval_dump() navíc i počet referencí. Funkce var_export() vypíše proměnnou tak, že je ji možné použít rovnu v PHP.

PHP nám přináší i magické konstanty __LINE__ (aktuální řádek), __FILE__ (aktuální soubor), __DIR__ (adresář aktuálního souboru),__CLASS__ (aktuální třída), __METHOD__ (aktuální metoda), __FUNCTION__ (aktuální metoda). Programátoři si spolu s výše uvedenými

magickými konstantami a funkcemi vytváří obvykle svá jednoduchá řešení, která umí například i logovat výpisy proměnných do souboru.

Příklad na magické konstanty:

```
function fce() {
    echo "soubor: " . __FILE__ .
    "\nřádka: " . __LINE__ .
    "\nfunkce: " . __FUNCTION__;
}
fce();
```

Výsledek pak může vypadat takto:

```
soubor: C:\xampp\htdocs\ODZ\K3_PHP\index.php
řádka: 5
funkce: fce
```

Potlačit chyby můžeme pomocí @-operátoru, který je možné psát před výrazy. Podívejme se na příklad s chybami, které však nejsou vypsané.

```
$pole = [1,2,3];
echo @$pole[100]; // žádný Notice: Undefined offset: 100
$concent = @file_get_contents('soubor.txt'); // žádný Warning: No such file or directory
```

Velmi důležitá je funkce error_reporting(), která nastavuje, jaké typy chyb chceme vypisovat. Pro vývoj, doporučuji mít nastavené E_ALL. Jinak máme možnost povolovat, či zakazovat poznámky (E_NOTICE), upozornění (E_WARNING), chyby za běhu (E_ERROR) a další. Na serverech v produkčním režimu jsou obvykle vypnuté poznámky a upozornění.

PHP umožňuje ošetřování vzniklých chyb i za pomocí výjimek (Exception). Výjimky jsou objekty, a proto je tomuto tématu věnováno více v kapitole Výjimky.

3.1.6 Shrnutí jazyka PHP

V této kapitole jsme se poměrně podrobně seznámili s jazykem PHP, který slouží obvykle ke generování webového obsahu, zejména pak v podobě HTML. Jelikož této kapitole předcházely klientské technologie HTML a CSS, tak by měl nyní čtenář být schopen samostatně vytvořit kvalitní dynamický web či jiné webové skripty dle zadání. Pro větší webové systémy doporučuji dále nastudovat objektové programování a práci s databází.

3.2 SQL

Jazyk SQL vznikl již v sedmdesátých letech minulého století, tedy velmi brzy po RMD (Relační model dat, The relational model), se kterým pracuje. Jedná se o dotazovací jazyk, který slouží pro komunikaci s databází prostřednictvím SŘBD. Existuje mnoho SŘBD pracujících s relačními databázemi podporující jazyk SQL. Pro příklad si uveďme: MySQL, MariaDB, Oracle, PostgreSQL, Microsoft SQL Server, SQLite, ale i Microsoft Access.

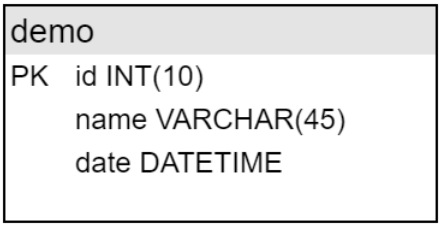
Jednotlivé příkazy jazyka SQL dělíme do určitých oblastí dle užití. **DCL** (***Data Control Language***) jsou příkazy pro management SŘBD a jejich uživatelů. Spadají sem například příkazy `GRANT` a `REVOKE`. **TCL**(***Transaction Control Language***) jsou příkazy pro řízení transakcí, kde příkladem jsou `COMMIT` a `ROLLBACK`. **DDL** (***Data Definition Language***) jsou příkazy pro definici struktur a jako poslední **DML** (***Data Manipulation Language***) jsou příkazy pro manipulaci s daty. Zejména tyto dvě poslední kategorie jsou často využívané programátory webových aplikací, kteří musí nejprve struktury (databáze, tabulky) vytvořit (`CREATE`), upravit (`ALTER`) a případně zrušit (`DROP`). Vytvoří-li se databáze a v ní tabulky, manipulujeme již s daty v tabulkách. Záznamy můžeme přidávat (`INSERT`), upravovat (`UPDATE`), mazat (`DELETE`) a samozřejmě velmi sofistikovaně získávat (`SELECT`).

Téma jazyka SQL je velmi široké a bylo na něj vypracováno mnoho materiálů, proto se zde nebudeme zabývat jednotlivými dotazy a strukturou jazyka.

PHP podporuje mnoho SŘBD (dBase, Firebird, IBM DB2, Informix, mSQL, MSSQL, MySQL, MariaDB, Paradox, PostgreSQL, SQLite, Sybase a další) a abstraktních vrstev (DBA, dbx, ODBC, PDO).

Pro databáze běžící na MySQL nebo MariaDB doporučuji rozšíření MySQLi nebo PDO_MySQL. Základy si ukážeme na jednoduchém příkladu. Mějme databázi s názvem *k5pr05* a v ní jednu tabulku *demo*. Databáze běží na vašem počítači, adresa je tedy localhost nebo 127.0.0.1 – 127.255.255.255. Přihlašovací jméno a heslo necháme výchozí. Jméno je *root* a heslo není zadané. Tabulka `demo` má tři sloupce. První je celočíselný primární klíč `id`, druhým je řetězec `name` a posledním je `date`. SQL generační skript vypadá takto:

```
CREATE TABLE IF NOT EXISTS `demo` (
`id` int(10) unsigned NOT null,
  `name` varchar(45) NOT null,
  `date` datetime DEFAULT null
) ENGINE=InnoDB AUTO_INCREMENT=18 DEFAULT CHARSET=utf8;
```



Obrázek 8 E-R schéma databáze pro testy v následných kapitolách

Rozšíření MySQLi podporuje přístup objektový i procedurální. Nyní si na jednoduchém příkladu výpisu řádek z tabulky `demo` ukážeme základní procedurální přístup:

```
$mysqli = mysqli_connect("localhost", "root", "", "k5pr05");
```

```
if (mysqli_connect_errno()) {
    printf("Chyba spojení: %s\n", mysqli_connect_error());
    exit;
}
$result = mysqli_query($mysqli, "SELECT *    FROM demo");
if (mysqli_errno($mysqli)) {
    printf("Chyba dotazu: %s\n", mysqli_error($mysqli));
    exit;
}
if (mysqli_num_rows($result) === 0) {
    printf("Tabulka je prázdná.");
    exit;
}
while ($row = mysqli_fetch_assoc($result)) {
    printf("%s\t%s\t%s\n", $row["id"], $row["name"], $row["date"]);
}
mysqli_close($mysqli);
```

Z příkladu vidíme, že nejprve se k databázi musíme připojit. Při úspěšném spojení pošleme do databáze SQL dotaz SELECT. Poté opět testujeme úspěch provedení dotazu, a navíc, i zdali není výsledek prázdný. Poté již jen řádek po řádku vypíšeme výsledek a spojení do databáze uzavřeme.

Výsledek by měl dle dat v databázi vypadat podobně tomuto:

```
15      Item    2018-12-08 16:55:43
16      Item    2018-12-08 17:33:01
17      Item    2018-12-08 17:33:12
```

Stejný příklad by v podobě objektového přístupu vypadal takto:

```
$mysqli = new mysqli("localhost", "root", "", "k5pr05");
if ($mysqli->connect_errno) {
    printf("Chyba spojení: %s\n", $mysqli->connect_error);
    exit;
}
if (!$result = $mysqli->query("SELECT * FROM demo")) {
    printf("Chyba dotazu: %s\n", $mysqli->error);
    exit;
}
if ($result->num_rows === 0) {
    printf("Tabulka je prázdná.");
    exit;
}
while ($row = $result->fetch_assoc()) {
    printf("%s\t%s\t%s\n", $row["id"], $row["name"], $row["date"]);
}
$mysqli->close();
```

Podrobněji si OOP vysvětlíme v další kapitole.

4 Objektově orientované programování

Předpokládám, že jste již pojem OOP někdy slyšeli. O objektech v souvislosti s programováním se mluví již více jak šedesát let. PHP není nativně objektový jazyk, ale práce s objekty byla do PHP přidána koncem devadesátých let ve verzi 3. Od verze 5 byla podpora OOP posílena a programátoři měli již poměrně dobrý standard pro psaní objektových aplikací. Postupně se objektový přístup zlepšoval, a i poslední verze 7 přinesla opět další pozitivní změny a možnosti.

OOP nám do programování může přinést škálovatelnost, přehlednost, srozumitelnost. Díky OOP můžeme udržet srozumitelné a přehledné i velké projekty. Pokud budete o objektovém přístupu číst v odborné literatuře, jistě se dočtete i o dědičnosti, polymorfismu nebo zapouzdření. Pojďme si to ale vysvětlit od začátku, postupně a jednoduše na příkladech. Poté by nám všechny výše uvedené pojmy měly být jasnější.

OOP vychází z našeho běžného života, ve kterém se běžně setkáváme s objekty. Nijak nás neudivuje, že jednotlivé objekty reálného života mají své stavy a také možnosti, jak tyto stavy měnit. Například dveře mohou být otevřené, zavřené, odemčené, zamčené a my máme možnost tyto stavy měnit. Je pro nás připraveno rozhraní, které nám je známé, v podobě zámku či kliky. Jak je mechanismus uvnitř zámku vyroben a jak funguje nás ani zajímat nemusí, abychom objekt mohli používat. Kdo ví, jak přesně funguje převodovka, brzdy nebo tachometr automobilu, a přeci jej milióny lidí používá. Také nás neudivuje, že se jednotlivé objekty skládají z jiných objektů anebo že některé objekty z jiných vycházejí, mají společné vlastnosti. Jednotlivé skutečné objekty vznikají dle patřičných postupů (receptů, „blueprintů“…) a mají tedy předdefinované společné vlastnosti a chování.

4.1 Třídy a objekty

Předpisy pro objekty se v programování nazývají třídy a vytváří se klíčovým slovem class, za kterým následuje název třídy a poté její tělo, které uzavíráme do složených závorek:

```
class Demo{}
```

Tělo třídy obvykle obsahuje atributy a metody. Atributy (vlastnosti) jsou nositeli stavu objektu a jsou reprezentovány proměnnými. Metody pak obsahují kód, který může stav objektu měnit, a jsou reprezentovány funkcemi. Vytvořme si veřejně přístupnou instanční proměnou p a veřejně přístupnou metodu `m()`, která obsah instanční proměnné vypíše:

```
class Demo {
    var $p;
    function m() {
        echo $this->p;
    }
}
```

Z třídy se tvoří konkrétní instance, kterým říkáme objekty. Každý objekt ze třídy `Demo` bude mít vlastní proměnnou `p` a tyto proměnné mohou nabývat různých na sobě nezávislých hodnot. V proměnné `this`, kterou používáme v metodě `m()`, je uložena reference na aktuální objekt, ze kterého metodu voláme. Tím můžeme psát těla metody univerzálně, a přitom metoda ovlivňuje atribut konkrétní instance třídy (objektu).

Z této třídy pak můžeme vytvářet objekty pomocí klíčového slova `new`. Referenci nově vytvořeného objektu pak jednoduše uložíme do proměnné `o`. Tomuto nově vytvořenému objektu nastavíme jeho proměnnou `p` a zavoláme jeho metodu `m()`:

```
$o = new Demo();
$o->p = 1;
$o->m();
```

V proměnné `o` je uložena reference na nově vytvořený objekt. Když tedy tuto proměnnou přiřadíme jiné proměnné, nevytvoříme tím kopii objektu, ale budou obě proměnné ukazovat na jeden objekt. Když tedy jednu z nich přepíšeme nebo zrušíme, objekt se tím nezruší, protože na něj stále existuje alespoň jedna reference:

```
$o = new Demo();
$o->p = 1;
$o->m();
$p = $o;
$p->p = 2;
$o->m();
$o = null;
$p->m();
```

Výstup:

```
122
```

Pokud budeme potřebovat kopii objektu, použijeme příkaz `clone`:

```
$o = new Demo();
$o->p = 1;
$p = clone $o;
$p->p = 2;
$o->m();
$p->m();
```

Výstup:

```
12
```

Ve výše uvedených příkladech je potřeba také vysvětlit šipku (pomlčka následovaná znaménkem větší než), která se tam za názvy proměnných často uvádí. Pomocí této šipky přistupujeme u konkrétních instancí (objektů, které jsou zastoupeny referenční proměnnou) k jejich patřičným atributům (proměnným) nebo metodám (funkcím). Tato šipka se používá vně i uvnitř třídy. V jiných objektově orientovaných jazycích se k tomu-

to účelu používá obvykle znak tečka, která však v PHP slouží ke spojování řetězců, a proto ji není možné použít pro referenci na atributy a metody.

Někdy potřebujeme objekty porovnat. K porovnání shody hodnot máme běžně operace == nebo ===, které můžeme použít i u objektů. Pokud objekty porovnáváme ==, tak jsou shodné, vyhoví-li všechny jejich atributy porovnání == (shodné hodnoty) a oba objekty jsou instancemi jedné třídy. Pokud objekty porovnáváme ===, tak jsou shodné, jedná-li se o dvě různé reference na stejný objekt. Podívejme se na malý příklad:

```
class A {
    public $x;
}
class B {
    public $x;
}

$o = new A();
$o->x = 1;
$p = $o;
var_dump($o == $p); //true
var_dump($o === $p); //true
$q = new A();
$q->x = 1;
var_dump($o == $q); //true
var_dump($o === $q); //false
$q->x = '1';
var_dump($o == $q); //true
$r = new B();
$r->x = 1;
var_dump($o == $r); //false
```

4.2 Instanční versus třídní prvky a konstanty

Atributy (proměnné) mohou být instanční nebo třídní. Každý objekt má své vlastní hodnoty instančních atributů. Přístup k němu je jen skrze objekt. Oproti tomu třídní proměnné nepatří objektu, nýbrž třídě. Přistupovat k nim můžeme jak skrze objekty, tak přímo pomocí názvu třídy. Druhá volba je však korektnější. Instanční atributy tedy patří objektu a třídní atributy pak třídě. Ve zdrojovém kódu je od sebe rozlišíme jednoduše, třídní proměnné mají před svým názvem klíčové slovo `static`.

U metod je to podobné. Před názvem třídní metody použijeme klíčové slovo `static`. Takto označené metody mohou pracovat jen se statickými proměnnými. Je to logické. Jak bychom u statické metody, kterou voláme pomocí názvu třídy, rozlišili zrovna tu konkrétní proměnnou té správné instance. Z této třídy navíc nemusí ještě žádný objekt existovat, tedy neexistuje ani instanční proměnná. Běžné metody mohou pracovat jak s instančními, tak i s třídními (statickými) atributy. K instančním atributům a metodám, které náleží konkrétnímu objektu, se uvnitř tříd přistupuje pomocí speciální proměnné `$this` následované šipkou. Ke statickým proměnným a statickým metodám se uvnitř třídy přistupuje pomocí klíčového slova `self` následovaného dvojitou dvojtečkou (`::`).

Konstantu můžeme chápat jako námi definovanou proměnnou, u které ale nesmíme měnit za chodu programu její hodnotu. Konstanta se tedy deklaruje podobně jako proměnné v úvodu třídy. Při deklaraci se provádí i její inicializace. Inicializace může být výraz, jehož hodnoty jsou předem známé. Ve výrazu není možné použít proměnné nebo funkce. Před názvem konstanty, který se obvykle píše velkými písmeny abecedy a pokud obsahuje více slov, tak se odděluje podtržítkem, se uvádí klíčové slovo `const`. V třídách se tedy nepoužívá funkce `define()`, kterou známe pro definici konstant v PHP u neobjektového přístupu. Ke konstantám se přistupuje jako ke statickému obsahu, tedy pomocí dvojité dvojtečky.

Příklad:

```
define("KONSTANTA", 1);

class Demo {
    public static $x = 3;
    const A = 2;
    const B = KONSTANTA + self::A;
    const C = self::B . '000';
    //const D = self::$x + 1; // není možné použít proměnnou i když je statická
}
var_dump(KONSTANTA, Demo::A, Demo::B, Demo::C);
```

Výstup:

```
int(1) int(2) int(3) string(4) "3000"
```

Tabulka 19 Způsoby přístupu ke statickému a instančnímu obsahu uvnitř a vně třídy

Prvek	Přístup uvnitř třídy	Přístup vně třídy
var p	$this->p	$instance->p
static p	self::$p	NázevTřídy::$p
function fce(){}	$this->fce()	$instance->fce()
static function fce(){}	self::fce()	NázevTřídy::fce()
const KONSTANTA	self::KONSTANTA	NázevTřídy::KONSTANTA

Kromě `$this->` a `self::` máme ještě `parent::`, který se používá k vynucení volání prvku rodičovské třídy u potomka. Toto si podrobněji ukážeme v kapitolce vysvětlující dědění.

4.3 Dědění (generalizace) a abstraktní třídy

U tříd, které mají významově shodné atributy nebo metody, bychom mohli pochybovat o jejich optimální struktuře. Také to můžeme chápat jako prohřešek proti DRY. Pravděpodobně bychom ji mohli najít společného předka. Příkladem nám může být třeba třídy `Customer` (zákazník) a `Employee` (zaměstnanec), které mají společné id, přihlašovací jméno a heslo. Naopak zákazník má předplatné oproti zaměstnanci, který má plat:

```
class Customer {
    var $id;
    var $username;
```

```
    var $password;
    var $subscription;
}

class Employee {
    var $id;
    var $username;
    var $password;
    var $pay;
}
```

Společný předek těchto tříd by mohla být třída `Person` s atributy `id`, `username` a `password`. Třídy `Customer` a `Employee` budou tyto atributy dědit, a navíc budou mít své vlastní atributy. Dědění zapisujeme ve třídě potomka (child) hned za názvem třídy pomocí klíčového slova `extends`, za kterým uvedeme název rodičovské (parent) třídy:

```
class Person {
    var $id;
    var $username;
    var $password;
}

class Customer extends Person {
    var $subscription;
}

class Employee extends Person {
    var $pay;
}
```

V potomcích `Customer` a `Employee` nyní můžeme používat atributy a metody, které jsou definované v rodičovské třídě `Person`. Potomek může dědit jen z jedné rodičovské třídy.

Z rodičovské třídy `Person` nebudeme chtít vytvářet instance. To zakážeme tak, že před klíčové slovo `class` u definice třídy uvedeme klíčové slovo `abstract`.

```
abstract class Person {
    var $id;
    var $username;
    var $password;
}
```

Nyní již není možné z třídy `Person` vytvářet instance, je označena jako abstraktní. Z jejich potomků to však možné je. Abstraktní mohou být i metody. U takto označených metod se definuje jen její signatura, nepíše se tělo. Jakmile třída obsahuje abstraktní metodu, sama musí být abstraktní. Abstraktní metoda musí být již ve svých potomcích definována i se svým tělem:

```
abstract class Person {
    var $id;
    var $username;
    var $password;
```

```
    abstract function show();
}

class Customer extends Person {
    var $subscription;
    function show() {
        echo "Customer id=" . $this->id;
    }
}

class Employee extends Person {
    var $pay;
    function show() {
        echo "Employee id=" . $this->id;
    }
}
```

Pro pochopení správného volání z potomků nebo rodičů jsem připravil jednoduchý příklad tříd, kde jsou tři varianty potomků, které nemají vždy všechny metody implementovány.

```
abstract class A {
    function callDump() { $this->dump(); }
    function dump() { var_dump(__METHOD__); }
}

class B extends A {
    function callDump() { $this->dump(); }
    function dump() { var_dump(__METHOD__); }
}

class C extends A {
    function callDump() { $this->dump(); }
}

class D extends A {
    function dump() { var_dump(__METHOD__); }
}

$b = new B();
$b->callDump();
$c = new C();
$c->callDump();
$d = new D();
$d->callDump();
Výstup:
string(7) "B::dump"
string(7) "A::dump"
string(7) "D::dump"
```

Všechny další možné varianty jsou pak znázorněny v následující tabulce 20. Volání metody pomocí `self::` a `parent::` v metodě potomka (child) rodičovské třídy (parent).

Tabulka 20 Volání metody pomocí self:: a parent:: v metodě potomka (child) rodičovské třídy (parent)

Volání v potomkovi	Metoda v potomkovi	Metoda v rodiči	Spustí se metoda
self::fce()	ANO	ANO	v potomku
parent::fce()	ANO	ANO	v rodiči
self::fce()	NE	NE	chyba
parent::fce()	NE	NE	chyba
self::fce()	ANO	NE	v potomkovi
parent::fce()	ANO	NE	chyba
self::fce()	NE	ANO	v rodiči
parent::fce()	NE	ANO	v rodiči

Mnohdy je dědění (generalizace) navrhováno tam, kam nepatří a pro dobrý návrh by byla lepší například kompozice. Pomoci nám může použití sloves „je“ nebo „má“ mezi třídou, která dědí a třídou, ze které se má dědit. Pokud je významově správnější sloveso „je“, pak má dědění (generalizace) svou logiku. V případě „má“ je vhodnější kompozice. Z našeho příkladu po překladu: „Zákazník“ je „Osoba“, dává větší smysl, než „Zákazník“ má „Osobu“. Sami se podívejte. Máme-li třídu Point:

```
class Point {
    private $x, $y;
    public function __construct($x, $y) {
        $this->x = $x;
        $this->y = $y;
    }
}
```

Pro třídu `Line` je lepší toto?

```
class Line extends Point {
    private $x2, $y2;
    public function __construct($x1, $y1, $x2, $y2) {
        parent::__construct($x1, $y1);
        $this->x2 = $x2;
        $this->y2 = $y2;
    }
}
```

Nebo je lepší toto?

```
class Line {
    private $startPoint, $endPoint;
    public function __construct(Point $startPoint, Point $endPoint) {
        $this->startPoint = $startPoint;
        $this->endPoint = $endPoint;
    }
}
```

4.4 Návěští viditelnosti

Protože jsme si již ukázali, jak se ve třídách používají atributy, konstanty a metody, a také již známe základní principy dědění, můžeme si vysvětlit návěští viditelnosti. Každému členu třídy můžeme určit, odkud k nim bude umožněn přístup. Možnosti máme `public`, `private` a `protected`, které se zapisují jako první u definice členu třídy. Public značí, že člen je veřejný a ke členu třídy můžeme přistupovat odkudkoliv. Toto nastavení je výchozí. Pokud tedy u metody nebo konstanty neuvedeme nic, nebo u atributu uvedeme `var`, pak tyto členy jsou automaticky veřejné (public). Private označuje soukromý člen, který je přístupný jen z třídy, ve které je definován. Protected se chová jako private, jen je umožněn přístup navíc ve všech potomcích, které z rodičovské třídy s protected prvkem přímo či nepřímo dědí.

Nastavení viditelnosti je u OOP klíčovou vlastností. Můžeme tak například zamezit přímý přístup k atributu, který nastavíme `private` a nastavovat jej jen pomocí vlastní `public` metody, kde si již korektní nastavení ošetříme. Proto jsou v OOP naprosto běžné metody, které atributy objektu nastavují (settery) a metody, které atributy objektu vrací (getery).

Rozšiřme si nyní náš příklad s osobami a nastavme si různé možnosti nastavení viditelnosti ke všem členům. Příklad stále není ideální, ale pro demonstraci poslouží dobře:

```
abstract class Person {
    private $id;
    public abstract function show();
    public function setId($id) {
        $this->id = intval($id);
    }
    protected function getId() {
        return $this->id;
    }
}

class Customer extends Person {
    private $subscription;
    public function getId() {
        return "c" . parent::getId();
    }
    public function show() {
        printf("id=%s, subs=%s\n", $this->getId(), $this->getSubscription());
    }
    public function setSubscription($subscription) {
        if (intval($subscription) >= 1 && intval($subscription) <= 12) {
            $this->subscription = intval($subscription);
        }
    }
    public function getSubscription() {
        if ($this->subscription != null) {
            return $this->subscription;
        } else {
            return 0;
```

```
        }
    }
}

$customer = new Customer();
$customer->setId(1);
$customer->setSubscription(10);
$customer->show();
```

Výsledek:

```
id=c1, subs=10
```

V našem příkladu je proměnná id soukromá a není možné ji použít jinde než ve třídě `Person`. Naštěstí třída `Person` disponuje veřejnou metodou `setId()`, která `id` korektně nastaví, a chráněnou metodou `getId()`, která jednoduše hodnotu `id` vrátí. Getter `getId()` je chráněný, a proto jej není možné volat jinde, než v metodách třídy `Person` a `Customer`. V potomku `Customer` poté přepisujeme metodu `getId()` tak, že si obstaráme `id` metodou `getId()` z rodiče `Person` a přidáme prefix „c“. Ve veřejné metodě `show()` již jen voláme patřičné gettery pro získání aktuálních hodnot atributů pro výpis. U veřejného setteru `setSubscription()` si ověříme, je-li v argumentu parametru `subscription` číslo v rozmezí 1–12 a pokud ano, hodnotu uložíme do atributu `subscription`. Veřejná metoda `getSubscription()` pak již atribut `subscription` vrátí, pokud není `null`. V opačném případě vrátí číslo nula.

4.5 Konstruktor a destruktor

V OOP jsou samozřejmostí konstruktory, tedy metody, které se automaticky volají při vytváření instance (objektu) ze třídy. Konstruktor slouží pro prvotní inicializaci objektu a jako funkce může obsahovat i parametry. Podobně existují i destruktory, tedy metody, které se automaticky volají při rušení objektu, tedy když na objekt neexistuje žádná reference. Destruktor nemůže mít parametry. V PHP se konstruktor nazývá `__construct()` a destruktor `__destruct()`. Obě tyto metody nic nevrací. Základní příklad na konstruktor a destruktor:

```
class Demo {

    private $id;
    private $name;
    private static $count = 0;

    public function __construct($name = "DemoName") {
        $this->id = ++self::$count;
        $this->name = $name;
        printf("Objekt %s vytvořen.\n", $this->id);
    }

    public function show() {
        printf("Objekt %s s názvem '%s'\n", $this->id, $this->name);
    }
```

```
    public static function getCount() {
        return self::$count;
    }

    public function __destruct() {
        self::$count--;
        printf("Objekt %s zrušen.\n", $this->id);
    }

}

printf("Počet objektů: %s\n", Demo::getCount());
$o = new Demo();
printf("Počet objektů: %s\n", Demo::getCount());
$p = new Demo("Druhý objekt");
printf("Počet objektů: %s\n", Demo::getCount());
$o->show();
$p->show();
unset($o, $p);
printf("Počet objektů: %s\n", Demo::getCount());
```

Výstup:

```
Počet objektů: 0
Objekt 1 vytvořen.
Počet objektů: 1
Objekt 2 vytvořen.
Počet objektů: 2
Objekt 1 s názvem 'DemoName'
Objekt 2 s názvem 'Druhý objekt'
Objekt 1 zrušen.
Objekt 2 zrušen.
Počet objektů: 0
```

V našem příkladu máme statickou proměnnou `count`, která se při každém vytvoření objektu zvýší o jednu a při každém rušení objektu sníží o jednu. V konstruktoru tím pak zajistíme, že každý objekt má své unikátní `id`. V konstruktoru je pak nepovinný parametr `name`, který v případě, že má argument hodnotu, ji přiřadí do atributu `name`, jinak se do atributu `name` uloží výchozí text „DemoName“.

Konstruktor a destruktor patří do tzv. magických metod. Zbylé magické metody si vysvětlíme v následující kapitole.

4.6 Další magické metody

Magické metody nejsou „magické“ tím, co dělají, protože jejich těla si píšeme sami, ale tím, že se volají automaticky při určitých událostech. Pokud tedy nastane patřičná událost, otestuje se existence potřebné magické metody ve třídě a v případě, že jsme ji v třídě definovali, tak se provede. Podobně se takto automaticky spouští konstruktor při vytváření nové instance z třídy. Magické metody začínají dvojitým podtržítkem. V následující tabulce je výpis magických metod s popisem vyvolávající události.

Tabulka 21 Přehled magických metod

Metoda	Výstup	Událost
__construct()	void	vytváření nové instance třídy
__destruct()	void	Na objekt neexistuje žádná reference.
__call()	mixed	volání neexistující/nepřístupné metody (objketový kontext)
__callStatic()	mixed	volání neexistující/nepřístupné třídní metody (statický kontext)
__get()	mixed	čtení hodnoty z neexistujícího/nepřístupného prvku
__set()	void	zápis hodnoty do neexistujícího/nepřístupného prvku
__isset()	bool	při volání funkce isset() nebo empty() na neexistující/nepřístupné prvky
__unset()	void	při volání funkce unset() na neexistující/nepřístupné prvky
__sleep()	array	požadavek na serializaci objektu funkcí serialize()
__wakeup()	void	požadavek na rekonstrukci objektu po serializaci funkcí unserialize()
__toString()	string	požadavek převést objekt na řetězec
__invoke()	mixed	volání objektu jako funkce
__set_state()	object	statická metoda volaná při exportu třídy funkcí var_export()
__clone()	void	vytváření klonu objektu
__debugInfo()	array	volání při převodu objektu na pole v argumantu funkce var_dump()

Pro názornost si uveďme příklad:

```
class Demo {

    private $data;
    private $x;

    public function __construct() {
        $this->data = array();
    }

    public function __set($name, $value) {
        if (strtolower($name) == 'x') {
            $this->x = $value;
        }
    }

    public function __get($name) {
        if (strtolower($name) == 'x') {
            return $this->x;
        }
    }

    public function __call($name, $arguments) {
        $method = substr($name, 0, 3);
        $attribute = substr($name, 3);
        if (!$method || !$attribute) {
            return;
        }
```

```
        switch ($method) {
            case 'set':
                $value = implode(', ', $arguments);
                if (!empty($value)) {
                    $this->data[$attribute] = $value;
                }
                break;
            case 'get':
                if (isset($this->data[$attribute])) {
                    return $this->data[$attribute];
                } else {
                    return null;
                }
        }
    }

    public function __invoke() {
        return count($this->data);
    }

    public function __toString() {
        return json_encode($this->data);
    }

}

$o = new Demo();// __construct()
$o->x = 5; // __set()
printf("%s\n", $o->x); //5 __get()
$o->setX(1); // __call()
$o->setY(2, 3, 4); // __call()
printf("%s\n", $o->getX()); //1 __call()
printf("%s\n", $o()); //2 __invoke()
printf("%s\n", $o); //{"X":"1","Y":"2, 3, 4"} __toString()
```

V komentářích jsou uvedeny magické metody, které se budou volat. U funkcí `printf()` je znázorněn i výstup. Pomocí magických metod `__set()` a `__get()` jsme si zpřístupnili privátní atribut `x`. Privátní atribut `data` zůstal nepřístupný. Pomocí magické metody `__cal()` jsme si však vytvořili mechanismus, který umožňuje použít univerzální settery a gettery a námi nastavované virtuální atributy ukládá do, respektive čte z, právě výše uvedeného privátního pole data. Pokud voláme objekt jako funkci, tak se zavolá metoda `__invoke()`, která vrátí počet prvků pole data. Pokud pracujeme s objektem jako s textovým řetězcem, zavolá se metoda `__toString()`, která pole data převede na **JSON** (JavaScript Object Notation), který záhy vrací.

Používání magických funkcí `__set()`, `__get()`, `__cal()` a `__callStatic()` se nazývá v PHP Přetěžování (**Overloading**). Všechny tyto metody musí být veřejné. Pozor na interpretaci pojmu Přetěžování u jiných programovacích jazyků, kde obvykle míní schopnost deklarace více metod se stejným názvem, ale rozdílnými parametry. V PHP se pojmem Přetěžování však myslí dynamický přístup k neexistujícím nebo nepřístupným prvkům třídy.

4.7 Rozhraní

Díky rozhraní vytváříme předpis, jak má třída navenek vypadat. Tedy pokud nějaká třída implementuje určité rozhraní, znamená to, že musí být všechny signatury metod v třídě stejné jako v rozhraní. Rozhraní se vytváří podobně jako třída. Jen místo klíčového slova `class` je použito `interface`. U třídy, která rozhraní implementuje, se pak uvede klíčové slovo `implements` a název rozhraní. Metody v rozhraní nemají svá těla a musí být veřejné, což je v souladu s pointou rozhraní. PHP nám také dovoluje programovat vůči rozhraní. Pro lepší pochopení si to představme tak, že pokud někde očekáváme patřičný objekt, nespecifikujeme jeho typ názvem třídy, nýbrž názvem rozhraní, které jeho třída implementuje. Oproti dědění, může jedna třída implementovat více rozhraní. Základní příklad:

```
interface IPerson {
    public function setId($id);
    public function getId();
}

interface ICustomer {
    public function setSubscription($subscription);
    public function getSubscription();
}

class Customer implements IPerson, ICustomer {
    private $id;
    private $subscription;
    public function setId($id) {
        $this->id = $id;
    }
    public function getId() {
        return $this->id;
    }
    public function setSubscription($subscription) {
        $this->subscription = $subscription;
    }
    public function getSubscription() {
        return $this->subscription;
    }
}
```

Ve výše uvedeném příkladu musí třída `Customer` implementovat oboje rozhraní, musí tedy implementovat metody `setId()`, `getId()`, `setSubscription()` a `getSubscription()`.

Rozhraní mohou dědit podobně jako třídy pomocí klíčového slova `extends`.

```
interface IPerson {
    public function setId($id);
    public function getId();
}

interface ICustomer extends IPerson{
    public function setSubscription($subscription);
    public function getSubscription();
```

```
}

abstract class Person implements IPerson {
    protected $id;
    public function setId($id) {
        $this->id = $id;
    }
}

class Customer extends Person implements ICustomer {
    private $subscription;
    public function getId() {
        return $this->id;
    }
    public function setSubscription($subscription) {
        $this->subscription = $subscription;
    }
    public function getSubscription() {
        return $this->subscription;
    }

}
```

V dalším příkladu si ukážeme programování vůči rozhraní a skládání objektů:

```
interface IEmployee {
    public function getId();
}

class Employee implements IEmployee {
    protected $id;
    public function __construct($id) {
        $this->id = $id;
    }
    public function getId() {
        return $this->id;
    }
    public function __toString() {
        return "Employee " . $this->getId();
    }
}

class Company {
    private $employees;
    public function addEmployee(IEmployee $employee) {
        $this->employees[] = $employee;
    }
    public function __toString() {
        return implode("\n", $this->employees);
    }
}

$company = new Company();
$company->addEmployee(new Employee(1));
```

```
$company->addEmployee(new Employee(2));
$company->addEmployee(new Employee(3));
echo $company;
Výstup:
Employee 1
Employee 2
Employee 3
```

Všimněme si, že metoda `addEmployee()` akceptuje do parametru objekt, který implementuje rozhraní `IEmployee`. Není zde tedy určeno, z jaké třídy musí být objekt vytvořen, ale jaký interface tato třída implementuje. Také tu máme další novinku, a to je skládání objektů. Tedy kdy jeden objekt může být složen z více jiných objektů. Zde je objekt z třídy `Company` tvořen několika objekty z třídy `Employee`. To je vytvořeno atributem `employees` v třídě `Company`, který je pole objektů z třídy `Employee`.

Rozhraní a abstraktní třídy jsou si v mnohém podobné. Oboje mohou nabídnout předpis pro jiné třídy. Zamyslete se, kdy je vhodnější použít právě interface a kdy abstraktní třídu.

4.8 Iterace objektu

Objekt lze iterovat atribut po atributu například pomocí řídící struktury `foreach`. Iterují se jen atributy, které jsou z místa volání viditelné:

```
class Demo {
    public $a = 1;
    private $b = 2;
    protected $c = 3;
    function iteruj() {
        echo "Vnitřní iterace:\n";
        foreach ($this as $key => $val) {
            printf("%s => %s\n", $key, $val);
        }
    }
}

$o = new Demo();
echo "Vnější iterace:\n";
foreach ($o as $key => $val) {
    printf("%s => %s\n", $key, $val);
}
$o->iteruj();
```

Výsledek:

```
Vnější iterace:
a => 1
Vnitřní iterace:
a => 1
b => 2
c => 3
```

4.9 Jmenné prostory

Díky jmenným prostorům si můžeme jednotlivé části kódu lépe zapouzdřovat. Jmenné prostory tvoří podobně jako adresářová struktura hierarchickou strukturu. Třídy ve stejném jmenném prostoru se nesmí jmenovat stejně. Jmenné prostory ovlivňují třídy, rozhraní, funkce a konstanty. Jmenné prostory se definují pomocí klíčového slova `namespace`, kde jeho první použití musí být jako první příkaz na začátku souboru. Před ním může být už jen `declare`. U pojmenování jmenných prostorů se nerozlišují velikosti písmen. Pomocí klíčového slova `use` si můžeme vytvářet aliasy.

Jednoduchý příklad:

```
namespace A\B\C;

class D {
    public static $x;
    public $y;
    public static function fceStatic() { var_dump(__METHOD__); }
    public function fce() { var_dump(__METHOD__); }
}

$o = new D();
$o->fce();
$p = new \A\B\C\D();
$p->fce();
D::fceStatic();

use A;
A\B\C\D::fceStatic();

use A\B\C;
C\D::fceStatic();
```

Výstup:

```
string(12) "A\B\C\D::fce"
string(12) "A\B\C\D::fce"
string(18) "A\B\C\D::fceStatic"
string(18) "A\B\C\D::fceStatic"
string(18) "A\B\C\D::fceStatic"
```

Z příkladu u vytváření objektu o vidíme, že nemusíme používat názvy jmenného prostoru, pokud jsme ve stejném jmenném prostoru jako deklarované třídy. U objektu `p` jsme již jmenný prostor uvedli. Nepřehlédněte však, že jsme určili i počátek pomocí lomítka. Pokud by tam lomítko nebylo, cesta jmenných prostorů by byla utvářena relativně od aktuálního. Hledala by se tedy třída `D` v prostoru `\A\B\C\A\B\C`. `Use` se používá typicky jako alias v nějakém jmenném prostoru na jiný jmenný prostor, abychom nemuseli stále psát odkazy na třídy, funkce, konstanty s kompletním určením jmenného prostoru.

4.10 Výjimky

Podobně jako je tomu i u jiných objektových jazyků, nabízí i PHP ošetření chyb s pomocí výjimek. Výjimky jsou objekty. Výjimka může být hozena (`throw`) a chycena (`catch`). Nezachycená výjimka je Fatal error: Uncaught exception. Výjimky „zkoušíme chytat" v `try` bloku, který musí mít minimálně jeden catch blok. V PHP je pro tento účel připravena přednastavená třída `Exception`, která nabízí mimo jiné metody `getMessage()`, `getCode()`, `getFile()` a `getLine()`.

Základní příklad zachycení vyhozené výjimky:

```
try {
    printf('Toto se provede.<br>');
    throw new Exception('Nastala chyba');
    printf('Toto se neprovede.<br>');
} catch (Exception $e) {
    printf("Zachycení výjimky: %s.<br>", $e->getMessage());
}
```

Výstup:

```
Toto se provede.
Zachycení výjimky: Nastala chyba.
```

Z výše uvedeného příkladu si ještě všimněte, že jakmile se v `try` bloku výjimka vyhodí, následné operace se již nevykonají a skočí se rovnou do `catch` bloků.

Nyní si ukážeme vyhazování výjimky ve funkci, vytvoření vlastní třídy pro výjimky a více `catch` bloků:

```
class DivisionException extends Exception {}

function divide($a, $b) {
    if (!$b) {
        throw new DivisionException("Division by zero");
    }
    $c = $a / $b;
    if ($c < 0.001) {
        throw new Exception("Result is too small");
    }
    return $c;
}

try {
    printf("Výsledek dělení: %f <br>", divide(1, 2000000));
    printf("Výsledek dělení: %f <br>", divide(1, 0));
} catch (DivisionException $e) {
    printf("Zachycení výjimky DivisionException: %s<br>", $e->getMessage());
} catch (Exception $e) {
    printf("Zachycení výjimky Exception: %s<br>", $e->getMessage());
}
```

Výstup:

```
Zachycení výjimky: Result is too small
```

Naše funkce `divide()` umí vyhazovat dva typy výjimek. Při dělení nulou vyhazuje naší vlastní `DivisionException`, která z `Exception` dědí a když je číslo malé, tak vyhazuje standardní `Exception`. Tato výjimka je tu nyní v této funkci jen pro demonstraci. V následném `try` bloku funkci pro dělení voláme s různými parametry a v případě vyhození výjimky ji patřičný `catch` blok ošetří.

Zkuste si zadávat různé hodnoty do parametrů funkce `divide()` v `try` bloku. Zkuste si `catch` bloky prohodit. Zkuste jeden, druhý, nebo oba `catch` bloky zrušit.

Nezapomeňme, že standardní knihovní vestavěné neobjektové funkce výjimky nevyhazují. Jednak nemá logiku, aby se toto dělo v neobjektovém kódu, a také by to zbouralo všechny již postavené webové aplikace.

4.11 Final

Nakonec si vysvětlíme klíčové slovo `final`, které se může použít u tříd a metod. Toto označení se používá před návěští viditelnosti. Pokud nastavíme třídu jako `final`, není z ní možné vytvářet potomky. Pokud nastavíme `final` metodě, není ji možné definovat (přepisovat) v potomcích. Toto dokonce platí i u `private` metod, které v potomcích není možné volat. Názorně si předveďme příklad, který znázorňuje chybné dědění z `final` třídy:

```
final class DemoFinal {}
class Demo extends DemoFinal {}
```

Výstup:

```
Fatal error: Class Demo may not inherit from final class (DemoFinal)
Předveďme si final i u metod:
class A {
    public static function x() {
        var_dump(__METHOD__);
    }
    final public static function y() {
        var_dump(__METHOD__);
    }
    final private static function z() {
        var_dump(__METHOD__);
    }
}

class B extends A {
    public static function x() {
        var_dump(__METHOD__);
    }
}

B::x(); // string(4) "B::x"
B::y(); // string(4) "A::y"
```

```
B::z(); // Fatal error:  Call to private method A::z() from context…
```

Ve třídě `B` můžeme přepsat jen metodu `x()`. Všechny ostatní díky návěští `final` přepsat nemůžeme. A to ani privátní metodu `z()`, která je z třídy `B` nedosažitelná.

4.12 Konvence, Best Practices

Pro přehlednost, srozumitelnost a udržitelnost kódu je dobré dodržovat dohodnuté konvence psaní kódu. Pro příklad uvedu několik doporučení:

1. Je dobré ctít strategie KISS, DRY a SOLID.
2. Krátké metody (cca 25 řádků) a krátké třídy (do cca 1000 řádků)
3. Vnořovat podmínky a cykly do hloubky maximálně cca tří úrovní.
4. Netestovat v podmínkách čistě jen proměnné, pokud nejsou typu bool. Používat isset(), empty()…
5. Řádně dokumentovat (DocBlock) zdrojový kód. Řádný komentář by měl být u každé třídy, rozhraní a jejich členů. Doporučovaný je například phpDoc.
6. Pro názvy proměnných, konstant, tříd, metod, souborů … používat anglická slova.
7. Názvy proměnných by měly být podstatná jména jednotného čísla, samo-popisné a loverCamelCase (například: $rowCount, $name). V případě polí či seznamů pak množná čísla.
8. Názvy metod by měla být slovesa, samo-popisné a opět loverCamelCase (například: setId(), renderText(), calculateRowCount()…).
9. Názvy tříd by měla být podstatná jména jednotného čísla v PascalCase.
10. Každá třída či interface je ve vlastním souboru s příponou php. Název souboru by měl být PascalCase.
11. Názvy souborů jsou case sensitive s dovolenými znaky a-zA-Z0-9-.

Pro PHP existuje například PSR-01 Basic Coding Standard a PSR-2: Coding Style Guide, ve kterých najdeme mnohá pravidla včetně těch, které jsme si výše již uvedli.

4.13 Shrnutí OOP v PHP

V PHP 5 je podpora OOP na dobré úrovni a s PHP 7 se to ještě zlepšilo. K dispozici zde máme třídy a jejich členy v podobě atributů a metod, návěští přístupnosti ke členům třídy, dědění, rozhraní, abstraktní třídy, velmi užitečné magické metody a mnohé další. PHP se snaží držet konvence, na které jsme zvyklí u jiných jazyků, i když se to někdy ne zcela povedlo. Příkladem je konstruktor.

Čtenář by po nastudování této kapitoly měl zvládnout navrhnout a implementovat v PHP program psaný dle pravidel objektově orientovaného programování. Tato kapitola byla jen vstupem do světa OOP v PHP. Čtenář by měl pokračovat dalším studiem této problematiky, naučit se psát dokumentační komentáře, studovat různé návrhové vzory, studovat jiná řešení a zejména programovat.

5 Seznam obrázků

6 Seznam tabulek