Lekce 3
(Teoretická část)

Dokumentace, komentáře v kódu

- Dokumentace je široký pojem, který může sahat od uživatelského manuálu až ke komentářům uvnitř kódu. Rozlišoval bych tyto typy od nejtechničtějšího:
    - 1) Komentáře v kódu
        - Asi každý imperativní programovací jazyk umožňuje psát do kódu vlastní komentáře. Komentáře se neřídí žádnými pravidly, je to prostě volný text, který jen musí být správně uvozený, aby program věděl, že se nejedná o kus kódu, který by byl určen ke zpracování.
        - Většinou máme možnost psát buď jednořádkové komentáře a nebo víceřádkový blok komentářového textu.
        - Příklad jednořádkového:

          // Toto je můj komentář, mohu si sem psát, co chci
        - Příklad víceřádkového:

          /* Toto je víceřádkový komentář.
             Mohu si psát do více řádků, což může být fajn. */

    - 2) Shrnutí metody, případně třídy. V php tzv. PHP docs. Jedná se o speciální komentář, který je strukturovaný a v nástrojích pro psaní kódu se potom ukazuje při volání dané metody, třídy. Do komentáře mohu specifikovat jaké argumenty jsou očekávané a co daná metoda asi dělá.
    - 3) Technická dokumentace kódu. Například dokumentace MSDN (příklad: https://docs.microsoft.com/en-us/dotnet/api/system.windows.documents.flowdocument?view=netframework-4.8) Zde se uplatňuje idea výše zmíněného **black boxu**. Dokumentace nepopisuje JAK daná třída/metoda funguje, ale jaké očekává argumenty, co vrací, jak se jmenuje a její základní použití. V případě tříd pak jaké má vlastnosti, metody atd.
    - 4) Technická dokumentace použití. Například dokumentace nějaké webové služby. Popisuje, jak má vypadat HTTP request, jaké hodnoty se očekávají.
    - 5) Uživatelská dokumentace. Je určena běžným uživatelům. Příkladem může být návod ke spotřebiči.
- Ke komentářům v kódu:
    - Platí, že není dobré psát ani příliš mnoho komentářů ani příliš málo. Obecně platí, že kód by měl být psán tak přímočaře a pojmenování metod, tříd a proměnných by mělo být tak srozumitelné, že komentářů není třeba. Jakmile napíšeme kus kódu, ve kterém je nějaká složitější nebo méně očekávaná logika, je vhodné napsat komentář. Pokud komentář chybí, může se velice dobře stát,

že někdo v budoucnu kód “opraví” takovým způsobem, že jej rozbije, protože netušil, proč byl kód napsán právě takto. Onou budoucí osobou můžeme být i sám autor kódu.

Funkce, procedury, metody a jejich parametry/argumenty

Rozlišování mezi pojmy: funkce, procedura, metoda

- Pojmy je víceméně možné zaměňovat. Jazyk PHP dokonce používá označení “function” pro něco, co by se správněji mělo nazývat “method”.
- Pokud mají tyto pojmy mít odlišné významy, pak by se od sebe lišily asi takto:
    - Funkce = vždy vrací nějakou hodnotu na základě zadaných argumentů.
        - Víceméně shodný význam jako v matematice
            - Mám funkci f(x) s předpisem = 2 * x
            - Když do funkce pošlu hodnotu 3, dostanu 6, když 4, dostanu 8, když 5, dostanu 10.
    - Procedura = může a nemusí mít parametry, nevrací žádnou hodnotu, jen něco provede.
        - V argumentu například pošlu objekt typu “kniha” a procedura provede zápis o doručené knize do databáze knihovny. Já již však nedostanu nic nazpět, žádnou zprávu ani hodnotu.
    - Metoda = kombinace předchozího. Může a nemusí mít parametry. Může a nemusí vracet nějakou hodnotu.
- Některé programovací jazyky mezi těmito pojmy rozlišuje, jiné nikoli. PHP ani C# pojmy nerozlišují. My zde také nebudeme rozlišovat.

Rozlišování mezi pojmy: parametr, argument

- Jedná se pouze o nuanci. Zde se budu snažit rozlišovat, ale běžně se pojmy zaměňují a z tohoto zaměňování většinou nevznikají žádná významná nedorozumění.
- Parametr se týká předpisu funkce. Říkáme například, že funkce má 3 parametry a z toho jeden nepovinný.
- Argument je konkrétní hodnota, kterou pro daný parametr posíláme. Pro stejnou funkci tedy mohu poslat pouze dva argumenty, protože třetí parametr je nepovinný.

Podstata metody (/funkce/procedury)

- Podstatou metody je deklarace předpisu a implementace obsahu (= kód, který se má vykonat).
- Předpis se skládá z deklarace:
    - Parametrů (jejich pořadí, typů, názvů)
    - Návratových hodnot (jejich typů)

        - Názvu metody
            - Název metody není zcela nutný, viz později tzv. “Anonymní funkce”
- Implementace
    - Je libovolným kusem kódu, který většinou nějakým způsobem zpracovává argumenty a případně navrací nějakou hodnotu jako výsledek zpracování.
- Plně explicitní zápis vypadá asi takto (C#):

```
Public string MyMethod(int parameter1, string parameter2) {
        // zde implementace
        string message = this.GetHelloMessage(“fr”);
        return message; // returns “Salut” třeba...
}
```

    - “public” je tzv. access modifier - o tom později
    - “string” deklaruje datový typ návratové hodnoty
    - “MyMethod” je identifikátor metody, jaký jsem si zvolil
    - “(int parameter1, string parameter2)” jsou parametry s explicitními datovými typy
    - Uvnitř {} následuje implementace metody zakončená klíčovým slovem “return”, které vrací hodnotu a musí odpovídat deklarovanému typu návratové hodnoty.
- Příklad metody v jazyce php (funguje v této podobě):
    - public function mojeMetoda($jmeno, $vek) {

```
        echo "Toto je " . $jmeno . "." . " Již oslavil(a) " . $vek . " narozenin.";
}
```

    - Když zavolám příkaz: mojeMetoda("Pavel", 25);
      Dostanu text: Toto je Pavel. Již oslavil(a) 25 narozenin.

<u>Deklarativní a imperativní přístup v případě metod</u>

- Na metodách je vidět kombinace dvou paradigmat: deklarativního a imperativního (viz dřívější lekce).
- Deklarativní paradigma v případě metod:
    - Při deklaraci metody nezáleží na jejím umístění od shora dolů. Imperativní “krokovátko” v případě deklarací nehraje žádnou roli. Například pokud metoda B závisí na metodě A (součást implementace metody), pak je stejně možné metodu B deklarovat “dříve”. Protože zde žádné “dříve”/”později” nedává smysl, pak je jedno, jestli je A zapsáno “před” B nebo naopak. Metody prostě jsou, jsou připravené k zavolání v určitém prostoru (= scope [o tom později]).
    - Metoda z pohledu deklarativního přístupu nemá žádnou implementaci. Deklaruje se pouze název, parametry a výstupní hodnoty.
        - Z deklarativního pohledu je metoda tzv. “**black boxem**”. Black box je v programování významným konceptem, protože deklaruje podmínky vstupním argumentů a slib návratových hodnot, přičemž implementace (= jak došlo ke zpracování vstupní dat), nás vůbec nezajímá. To je podstatné, protože implementace může být složitá a v mnoha úrovních

        zanořená, zatímco nás na vyšší úrovni tato složitost nemusí vůbec zajímat.
        - Z pohledu webového programování je celá aplikace takovým black boxem. Klient zašle HTTP request serveru. Tam se odehraje pro uživatele zcela neznámá logika a navrací se výsledek v podobě webové stránky.
        - Jiným příkladem je třeba situace ve fast foodu. Vstupními parametry jsou: dodané peníze a určení názvu pokrmu a výstupní hodnotou je pokrm. Mne jakožto “klienta” by nemělo vůbec zajímat, jak přesně se k výsledku došlo, pokud dostanu to, co jsem si objednal.
    - S čistě deklarativním zápisem metody se dokonce setkáme při reálném použití (php, C# a další). Jedná se například o deklaraci tzv. abstraktních metod nebo rozhraní (interface). Abstrakcemi se budeme podrobněji zabývat později. Idea je však stále stejná: deklaruje se jaká metoda někde má být a později určí její implementace.
- Imperativní paradigma v případě metod
    - Metody se v imperativním paradigmatu uplatňují jednak “zevnitř”, jednak “svrchu”.
    - Zevnitř:
        - = implementace metody. Vnitřek, implementace, metody je tradičním imperativním paradigmatem, kde má smysl “krokovátko”.
    - Svrchu:
        - = volání metody. Metoda je připravena k zavolání stejně jako jakýkoli “příkaz”. Volána je při exekuci jakmile k příkazu zavolání dojde krokovátko. V této chvíli se metodě předávají argumenty a metoda následně vrací hodnotu (nebo nikoli).

Princip definice kódu či hodnot na jednom místě

- V programování je téměř vždy chybou, pokud něco, **co je stejné** (či stejné do značné míry) **kopírujeme na více míst**. V takových případech je třeba se zamyslet a pokusit se situaci vyřešit tak, že kód či hodnotu nebudeme kopírovat, ale budeme pouze odkazovat na společné místo.

Analogie

- Analogie #1:
    - Představme si, že máme fotku z dovolené uloženou na svém PC, a protože si vytváříme domácí knihovnu, máme stejnou fotografii zkopírovanou jak ve složce “rodina”, tak ve složce “dovolené”. Stejnou fotku máme tedy dvakrát. Jednak nám fotka ubírá na disku dvojnásobné místo. Ale co je horší - pokud se rozhodneme

soubor přejmenovat a nebo třeba upravit barvy na fotce, pak musíme nalézt všechny její výskyty a provést úpravu také tam.

- Analogie #2:
  - Pokud spolupracujeme s více kolegy na stejném textu, je dobré mít sdílený dokument nebo alespoň sdílený adresář obsahující daný dokument. Pokud by každý z kolegů měl svou vlastní kopii dokumentu, pak by stejnou úpravu v mém textu museli provést také všichni ostatní, což vyvolává chaos.

Příklad z programování - funkce

- Potřebujeme například počítat libovolnou mocninu libovolného čísla.
- To lze vyřešit cyklem jako například:

```
$umocneneCislo = $puvodniCislo;
for($i = 1; $i < $mocnina; $i++) {
        $umocneneCislo = $umocneneCislo * $puvodniCislo;
}
```

- Mohli bychom chtít postupovat tak, že kdykoli by se vyskytla potřeba tohoto výpočtu někde v kódu, tak bychom zkopírovali tento kód, dosadili hodnoty a získali výsledek.
- **Tento postup by fungoval, ale je to velice špatná praxe.**
- Pokud bychom například zjistili, že v tomto kusu kódu máme nějakou chybu, pak bychom museli vyhledat všechna místa, kde jsme toto napsali a chybu všude opravit.
- Daleko lepší je tento kód zapsat jako funkci:

```
Public function mocnina($cislo, $mocnina) {
        $umocneneCislo = $puvodniCislo;
        for($i = 1; $i < $mocnina; $i++) {
                $umocneneCislo = $umocneneCislo * $puvodniCislo;
        }
        Return $umocneneCislo;
}
```

- Ze všech míst potom budeme volat pouze funkci: mocnina($cislo, $mocnina); a bude-li třeba kód upravit, **pak jej stačí upravit jednou, v definici funkce**.

Příklad z programování - hodnoty

- Podobně je tomu s hodnotami. Například chceme různě v kódu pracovat s kódem pro český jazyk. Například když se bude jednat o češtinu, napíšeme “Ahoj”, když o angličtinu, napíšeme “Hello”.
- Zkratku pro češtinu si například zvolíme “cz” a angličtinu “eng”.
- V kódu bychom pak na různých místech mohli začít chtít psát podobné kusy kódu:

```
If ($jazyk == “cs”) {
        echo “Ahoj”;
}
```

- **Tento postup by opět fungoval, ale opět se jedná o špatnou praxi**.

- Pokud bychom například chtěli změnit kód z “cz” na “cs”, museli bychom najít všechna místa, kde toto bylo použito.
- Daleko lepší je hodnotu někde jednou definovat a pak se na ní odkazovat. Jedna z možností je definovat tzv. **konstantu**, tedy neměnnou hodnotu. V php například napíšu:
  const JAZYK_CESTINA = ‘cz’;
  const JAZYK_ANGLICTINA = ‘eng’;
- Kdekoli potom budu potřebovat použít tento jednotný kód pro češtinu, tam se odkážu na tuto konstantu. Pokud pak bude potřeba kód změnit, stačí upravit hodnotu konstanty.

Číselníky

- S definicí hodnoty na jednom místě souvisí pojem “číselník”. Číselníky mohou mít mnoho podob, ale nejrozšířenější a nejmocnější je asi podoba číselníku uloženého v databázi (**o databázích později**).
- Základem číselníku je, že pro danou hodnotu je určeno neměnné ID (tradičně číslo, ale může se jednat o textový kód typu “ENG”), na které se pak všude odkazujeme.
- Výhoda číselníku uloženého v databázi má výhodu v tom, že daný číselník můžeme použít pro mnoho různých aplikací a ještě jej například můžeme zveřejnit pro externí aplikace.
- Příkladem velice obecného číselníku sdíleného mnoha aplikacemi ve světovém měřítku je třeba ISO 3166-1 (https://cs.wikipedia.org/wiki/ISO_3166-1), číselník států. Česká republika má zde kód 203 a pokud by všechny aplikace používali tento číselník, pak bude komunikace mezi nimi konzistentní. Všude, kam zašlu konstantu 203, budou všichni vědět, že se jedná o Českou Republiku. Oproti tomu zasílání něčeho jako “Czech Republic”, “Czechia”, “Česká Republika”, “ČR”, “CZ”, “CS”, “CR” by bylo velice matoucí a komunikace by byla obtížná.

Algoritmizace, pseudokód, výpočetní komplexita

Algoritmizace

- Vlastní nepřesná a záměrně obecná definice pojmu algoritmus = **definice po sobě jdoucích kroků**, kterými se člověk snaží získat odpověď (= řešení) na položenou otázku (= problém, situace).
- Algoritmus lze dle výše uvedené obecné definice vztahovat na libovolný problém či situaci. Kořeny ideje jsou však v matematice. Použití algoritmu v oblasti matematiky značí podstatně odlišný přístup k řešení matematického problému.
  - Mám například následující problém: x + 2 = 5, tedy “Pro jaké číslo platí, že jeho součet s číslem 2 bude se rovnat 5?” či prostě “Čemu se rovná x?”
    - Mohu řešit buď exaktním přístupem, analýzou problému, tedy:
      - x = 5 - 2, tedy x = 3

- Nebo mohu řešit algoritmicky a to všemožnými způsoby, například takto: Za x dosadím číslo 0 a zjistím, zda se levá strana rovnice rovná pravé. Pokud ano, mám odpověď. Pokud ne, pak k posledně hádanému číslu přičtu 1 (původně byla 0 a 0 + 1 máme 1) a opět porovnám a vyhodnotím, zda mám výsledek. Pokud nikoli, posledně hádané číslo vynásobím (-1), tedy otočím znaménko. Vyhodnotím. Pokud nemám výsledek, opět otočím znaménko a přičtu číslo 1. Opět vyhodnotím. Zde se již algoritmus točí dokola - stále zkouším obě znaménka hádaného čísla a po neúspěchu přičitám 1.
    - Touto metodou, postupem, bych dříve či později měl dojít k výsledku řešení problému a to algoritmickou metodou.
- V matematice algoritmy souvisí s pojmem “numerická analýza”, které je užíváno zejména v komplikovaných problémech, kde ani nemusí být známá (či ani možná?) exaktní metoda pro řešení problému. Výsledkem problému je často zjištění přibližné hodnoty, nikoli přesné odpovědi.

Pseudokód

- Pseudokód je zápis definice algoritmu syntakčními značkami, které jsou všeobecně srozumitelné, aniž by se zároveň muselo jednat o zápis, který by byl platný v jakémkoli programovacím jazyku.
- Pseudokód úzce souvisí s pojmem algoritmu a vlastně nám umožňuje algoritmus vyjádřit heslovitěji a přesněji, než by nám to umožňoval běžný jazyk s přidanou hodnotou podobnosti běžnějším programovacím jazykům.
- Zápis pseudokódu může vypadat třeba takto:

```
funkce(x : integer, y : integer) {
        If (x > y) then
                return true;
        Else
                return false;
}
```

- ⇒ co nám tento pseudokód říká?

Výpočetní komplexita algoritmu a “big O notation”

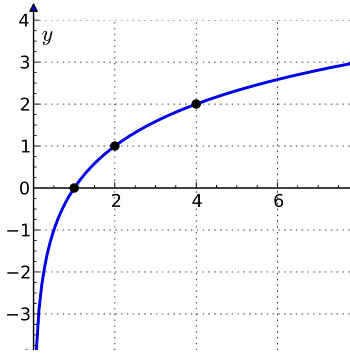
- Jeden problém mohu vyřešit mnoha různými algoritmy, mnoha tzv. Implementacemi.
- Jeden algoritmus může být efektivnější než jiný algoritmus, což lze při konstantní výpočetní síle stroje brát jednoduše jako rychlost, s jakou je problém vyřešen (čas, za který je vyřešen). Při porovnávání efektivity algoritmů ale hrají velkou roli vstupní hodnoty, tedy je třeba uvažovat o tom nejhorším možném případu, ale také o tom nejlepším a dokonce také o tom “běžném”.
- Příklad:

- Máme kolekci celých čísel. **Kolekce je seřazená** od nejmenšího čísla po největší. Čísla se nesmějí opakovat a kolekce má rozměr třeba 100 000 členů. Naším úkolem je zjistit, na jaké pozici se poprvé nachází číslo “5” a zda-li vůbec na nějaké.
- Asi nejhorší situací je, když se v kolekci číslo “5” vůbec nebude nacházet, protože nás nic nezastaví při nálezu.
- Snahy o řešení (#1):
    - Jednou z metod může být náhodné zkoušení pozic s tím, že si alespoň budu pamatovat, co jsem vyzkoušel. Nejlepším výsledkem zde může být, že se trefím hned napoprvé. Nejhorším výsledkem může být, že se trefím až na poslední odhad nebo že se dané číslo v kolekci vůbec nebude nacházet.
    - Další metodou může být, že budu kolekci procházet z jedné ze stran dokud číslo nenajdu nebo nedojdu konce kolekce.
    - ⇒ V obou případech je situace stejná. Co nás však zajímá je onen nejhorší scénář. V obou případech je nejhorším scénářem, že budu muset projít všechny prvky kolekce. Mám-li tedy “n” prvků, pak budu muset vykonat “n” porovnání čísel. S přibývajícím počtem prvků v kolekci tedy úměrně (= lineárně) přibývá počtu nutných operací.
- Řešení situace binárním vyhledáváním (#2)
    - Zvolím prostřední index kolekce. Je-li to hledané číslo, mám výsledek. Pokud je číslo větší, pak opakuji stejný postup pro pravou část rozdělené kolekce. Pokud je číslo menší, pak opakuji postup pro levou část rozdělené kolekce. Takto pak kolekci stále dokola dělím až se doberu k výsledku, je-li přítomen.
    - ⇒ Nejlepší možná situace je, že se trefím hned napoprvé, tedy to je stejné jako v předchozím případě. Opět nás ale zajímá nejhorší možná situace. A zde je významná odlišnost! Počet nutných operací je totiž tzv. logaritmus počtu prvků v kolekci. Tedy čím více budu mít členů kolekce, tím sice nejhorší možná situace bude také narůstat, ale již neúměrně vůči počtu členů, logaritmicky, tedy bude narůstat stále méně.
- ⇒ Když v prvním případě dostanu v kolekci miliardu prvků, budu jich muset v nejhorším možném případě miliardu projít. Zde hledání dělím v půli, takže hned v prvním kroku jsem vyloučil 500 milionů nutných operací. V dalším kroku 250 milionů a tak dále.

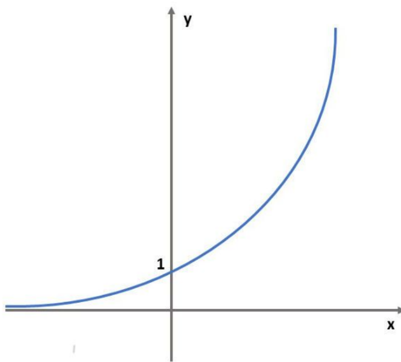
- Toto je příkladem odlišné efektivity obou algoritmů, tedy jejich komplexity. Komplexita se zapisuje tzv. Big O notací (spojením “big O” je míněno písmeno “O”, takže řecké Omicron).
    - V prvním případě je nejhorší možný případ tzv.: O(n).
    - V binárním vyhledávání je nejhorší možný případ: O(log n).
- Obecně rozlišujeme především mezi těmito komplexitami:
    - O(log n): logaritmická komplexita
    - O(n): lineární komplexita

- $O(n^k)$: kvadratická (či jiná mocnina)
- $O(e^x)$: exponenciální komplexita

Logaritmická řada



Exponenciální řada: $O(e^x)$



Lineární řada: $O(n)$