TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Java pro ztracené případy

Semestrální práce

Studijní program: B0613A140005 – Informační technologie
Studijní obor: B0613A140005IT – Informační technologie

Autor práce: **Kevin Daněk**
Vedoucí práce: -

Liberec 2022

Tento list nahraďte
originálem zadání.

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má semestrální práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

17. 5. 2022 Kevin Daněk

Java pro ztracené případy

Abstrakt

Práce se snaží jednoduchými metaforami a analogiemi uvést čtenáře do problematiky programování, konkrétně programování v objektově orientovaném jazyce Java.

Klíčová slova: Java, návod, programování, objektově orientované programování, základy programování

Java for lost causes

Abstract

This document is trying to explain fundamental programming concepts through simple analogies and metaphores reflecting the real life scenarios. It's main focus is explaining the problem of creating an algorithm and expressing it through structural and object oriented programming patterns.

Keywords: Java, guide, programming, object oriented programming, programming fundamentals

Poděkování

Děkuji lidem, co si tohle opravdu přečetli.

Obsah

1 Úvod **8**

2 Úvod do programování a algoritmizace **9**
2.1 Programování a algoritmizace 9
2.1.1 Asymptotická složitost algoritmů 10
2.2 Datové typy 12
2.3 Programovací paradigma 13
2.4 Strukturované programování 14
2.5 Datové struktury 14
2.6 Výjimky a chybové stavy 15

3 Úvod do objektového programování **16**
3.1 Objekty 16
3.1.1 Atributy 16
3.1.2 Metody 17
3.2 Vytváření objektů a přístup k objektům 17
3.2.1 Třída objektu 17
3.2.2 Konstruktor a destruktor 17
3.2.3 Modifikátory přístupu 18
3.3 Interakce mezi objekty 18
3.3.1 Abstrakce a zapouzdření 18
3.3.2 Skládání objektů 19
3.4 Statická deklarace 19
3.5 Dědičnost 19
3.6 Polymorfismus 20

4 Závěr **21**

Seznam literatury **22**

Přílohy **23**

A Slovník pojmů **23**

Seznam obrázků

2.1 Diagram tvorby programu 10
2.2 Graf asymptotických složitostí (freecodecamp.org) 11
2.3 Ukázka proložení bodů z měření parabolou 12

Seznam tabulek

2.1 Hlavní rysy strukturovaného programování 14

1 Úvod

Tato práce se zaobírá problematikou programování, a to konkrétně obecné algoritmizace a lehkého úvodu do OOP. Kvůli omezení rozsahu je odstraněna Java část a kus OOP části.

V textu nejsou jednotlivé pojmy explicitně definovány, proto jsou definice jednotně uvedeny ve slovníku, který naleznete v příloze.

2 Úvod do programování a algoritmizace

2.1 Programování a algoritmizace

Pod programováním si spousta lidí představí spoustu věcí - kdybych ale měl tento pojem popsat jednou větou, asi bych ho popsal nějak takto:

Ačkoliv né uplně přesná definice, podle mého názoru dává nejlepší představu o tom, co programování vlastně je. Programování je o hledání řešení nějakého problému, jak toto řešení vyjádřit a zajistit, aby řešení bylo co nejvhodnější danému problému. Programování už z výše uvedené definice říká, že vede k nějakému postupu, který daný problém řeší. Tento postu označujeme honosným názvem "**algoritmus**".

Nejčastěji se setkávám s přirovnáním ke kuchařce, proto si ho dovolím použít i zde: Za algoritmus můžeme považovat recept, kde máme přesnou sadu instrukcí, které nám říkají, jak něco uvařit. Tenhle recept vždy vede ke stejnému výsledku (nemůže se nám stát, že pomocí receptu na koprovku uděláme rajskou, že ano) a vždy bude doba vaření trvat stejně dlouho.

Už jenom z této intuice můžeme odvodit některé vlastnosti algoritmů:

1. **Obecnost -** Algoritmus by měl být dostatečně obecný na to, aby dokázal řešit pokud možno co nejvíce případů.
2. **Determinovanost -** Výsledek algoritmu by měl být předvídatelný.
3. **Konečnost -** Algoritmus musí v nějakém bodě skončit.
4. **Srozumitelnost -** Algoritmus by měl dávat smysl a nemělo by být zbytečně složité ho pochopit.
5. **Efektivita -** Algoritmus by neměl plýtvat zdroji a být pokud možno co nejúspornější. To samé platí i o čase potřebném na jeho vykonání.

Tato pětice bodů je kolektivně známa jako **vlastnostni algoritmu**. Ovšem jak takový algoritmus vytvoříme? Proces takzvané **algoritmizace** by se dal popsat v následujících krocích:

1. **Analýza zadání -** Pořádně se podíváme, co se po nás a našem algoritmu chce.
2. **Rozložení na menší problémy -** Pokud je původní zadání komplexnější, rozložíme si ho na několik menších problémů, které budeme postupně řešit

3. **Návrh řešení -** Rozmyslíme, jakými způsoby můžeme dílčí problémy řešit, a který postup by byl pro nás nejlepší.

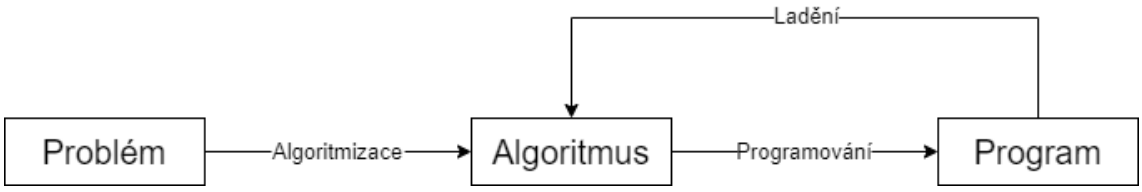
4. **Zápis algoritmu -** Algoritmus zformulujeme a zapíšeme

5. **Ladění -** Algoritmus zkontrolujeme, zda se chová tak, jak má.

Výsledkem této algoritmizace bude konečný sled kroků, který můžeme následně použít k vytvoření programu. Jak ale takový algoritmus zapíšeme?

Metod je spousta, a jsou to asi takové, které by vás napadly. Můžeme algoritmus vyjádřit slovně, písemně po větách, v programovacím jazyce nebo pomocí vývojových diagramů. Hlavně poslední způsob zápisu je velmi oblíbený ve školách na otravování studentů, proto se ním tady zabývat nebudu.

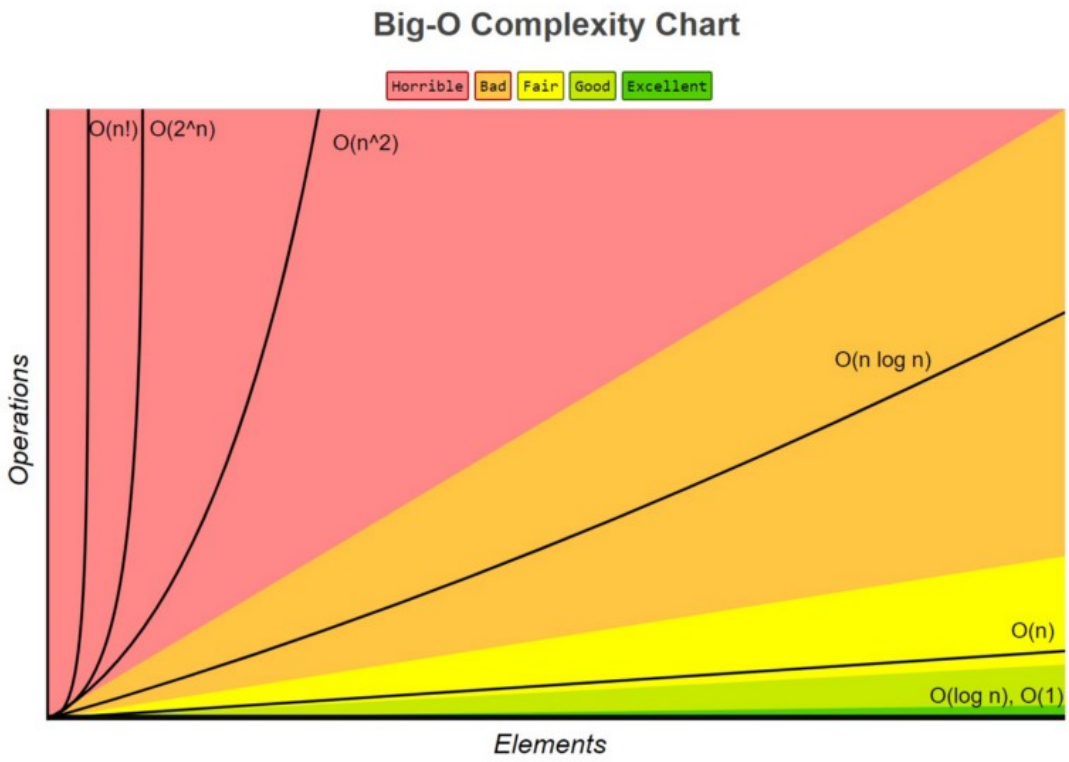
Na závěr téhle kapitoly bych vám rád ukázal diagram níže, který propojuje všechny výše zmíněné pojmy dohromady a jak vlastně spolu souvisí:



Obrázek 2.1: Diagram tvorby programu

2.1.1 Asymptotická složitost algoritmů

Vzpomeňte si zpátky na **vlastnosti algoritmu** - jednou z nich byla **efektivita**. Jak ale určíme, co je "dostatečně dobrá" efektivita, a co ne? Ačkoliv to nejde přímo univerzálně určit, protože to vždy záleží na dané situaci, ale z matematiky si můžeme vypůjčit nástroj, který nám dovolí zhruba určit, jak dlouho bude nějaká operace trvat. Tomuto nástroji se říká **asymptotická složitost** nebo také **Big O Notation**.



Obrázek 2.2: Graf asymptotických složitostí (freecodecamp.org)

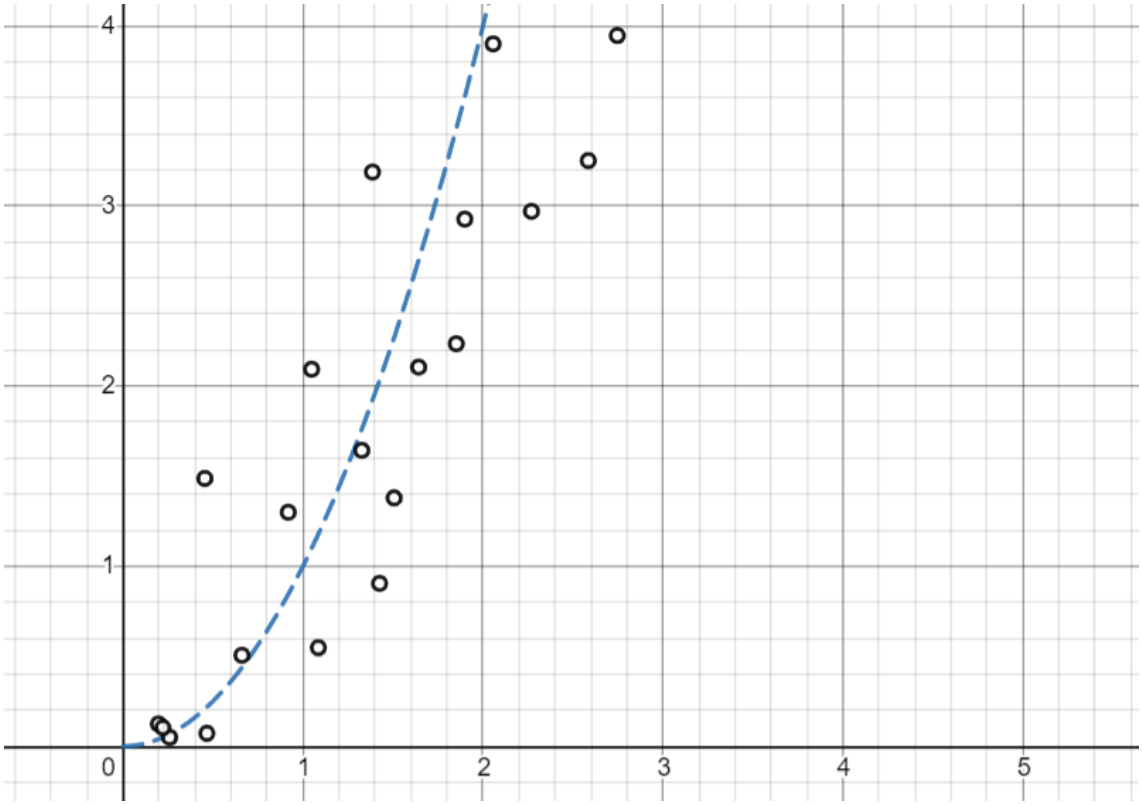
Asymptotická složitost se určuje hlavně pro dvě domény:

1. **Časová složitost -** Jak rychle roste čas potřebný na zpracování dat
2. **Paměťová složitost -** Jak rychle roste paměť potřebná na provedení algoritmu

Málo kdy se podaří mít optimalizovat jak čas, tak paměť (zpravidla bývá optimalizace jedné veličiny na úkor druhé). V dnešní době se spíše upřednostňuje optimalizace času, jelikož dostupné paměti je dnes docela dost.

Jak tedy takovou časovou asymptotickou složitost vypočítat? No, to je trochu složitější a je důležité říct, že je většinou spíše o odhady. Typ složitosti se dá zpravidla odhadnout z konkrétního kódu (například dvě vnořené smyčky do sebe mívají kvadratickou složitost).

Pokud to ale nelze odhadnout, lze samozřejmě provést měření a určit jednu z funkcí, která nejlépe odpovídá nárustu v čase. Podívejte se na ilustrační obrázek níže - Na ose x máme počet vstupních dat (z pravidla se používá logaritmické měřítko) a na ose y čas, který byl potřeba na vykonání.



Obrázek 2.3: Ukázka proložení bodů z měření parabolou

Černé kroužky jsou body, které odpovídají naměřeným hodnotám, a modrá přerušovaná čára je pravá větev paraboly, kterou data prokládáme. Když se na takhle na tento obrázek podíváme, asi se shodneme, že hodnota bodů roste tak nějak podobně jako parabola, kterou jsme přidali.

S tímto závěrem bychom řekli, že funkce, která vytvářela tyhle body, má **kvadratickou časovou složitost**, nebo bychom zapsali v podobě O-notace $O(n^2)$.

Důležité si je uvědomit, že nám to neříká nic jiného, než jak čas roste v závislosti na objemu vstupních dat - nemá to být nějaký přesný výpočet - spíše jenom pomůcka k rychlému ohodnocení efektivity algoritmu.

2.2 Datové typy

Při programování pracujeme s daty různých typů - můžeme pracovat s celými čísly, reálnými čísly, s řetězci znaků a dalšími blbostmi. Datové typy programovacímu jazyku říkají, co za data chceme v proměnné ukládat, a respektive, jak je má ukládat v paměti.

Nejčastější a nejzákladnější datové typy jsou celá čísla (*integers*), reálná čísla (*doubles*), znaky (*chars*) a textové řetězce (*strings*).

Občas se ale může stát, že máme data uložena v jednom datovém typu, například řetězci znaků, a potřebujeme s ním pracovat jako s jiným typem, například číslem.

To se stává často u zpracování uživatelského vstupu, který bývá celý uložen v řetězci znaků (*string*) [2]. Když chceme po uživateli zadat čísla, dostaneme akorát znaky, a my si musíme poradit. Pokud chceme z řetězce znaků dělat číslo, přichází na řadu proces **přetypování**.

Přetypování bývá velmi nebezpečný proces, který je často náchylný na vyvolání chyby. Proč? Představme si opět převod řetězce znaků na číslo. Při převodu mohou nastat 3 případy:

1. V řetězci jsou jenom cifry, tudíž bude převod úspěšný
2. V řetězci jsou cifry a nějaký další znak, s trochou štěstí dostameme akorát to číslo, v horším případě chybu
3. V řetězci jsou znaky, které nelze přeložit na číslo. Buďto se vyvolá chyba, nebo se nám vrátí hodnota typu NaN.

To jsou takové základní případy, které by mohly nastat. Může jich být více či méně, hodně záleží na samotných programovacích jazycích, jak moc se snaží běh programu zachránit. Některé při převodech chybu nevyvolají, ale pak musíme kontrolovat správnost převodu.

2.3 Programovací paradigma

Když vysvětluji, co je programovací paradigma svým kamarádům, rád říkám, že to je takový "styl programování" a přirovnávám to k zaklínačským školám z povídek Andrzeje Sapkowského (a respektive jejich videoherním adaptacím).

Každá zaklínačská škola učí své zaklínače, jak se připravit na souboj, jak získat výhodu v boji a jak vůbec bojovat - zda být hbitý a mít výhodu rychlosti, či si držet půdu pod nohama a získat výhodu síly. Podobně je to i s programovacími paradigmaty, kde by každé paradigma odpovídalo bojovému stylu jiné zaklínačské školy.

Každé paradigma (styl) programování popisuje problém jinak, a tak vždy k výsledku dospěje jinou cestou, obecně je ale můžeme rozdělit na dvě velké kategorie:

1. **Imperativní -** Zde říkáme, jaké kroky má program udělat, abychom dospěli ke chtěnému výsledku
2. **Deklarativní -** Zde zase říkáme, co za výsledek chceme, a necháme program, ať se k němu nějak dopracuje.

Nejčastěji se asi setkáte s jazyky **imperativními**, mezi které patří velká jména jako *Java, C, C++, C#, Rust, Python* a spousta dalších. Mezi **deklarativními** jazyky jsou také velká jména, ale jsou to jazyky spíše pro kontrolu nějakých rozhraní (například *SQL* pro vytváření dotazů na databázi, či *CSS* pro stylování webové stránky).

2.4 Strukturované programování

Strukturované programování je označení pro paradigma, které vzniklo z frustrace nad nestrukturovaným programováním, ve kterém bylo složité větvit program podle aktuálního stavu dat či opakovaně volat nějaký blok kódu. Ačkoliv je toto knížka o Javě, která je objektově orientovaná, připadá mi dobré zmínit základní struktury (viz tabulka 2.1), na kterých strukturované programování stojí, jelikož je objektově orientované programování v jistém slova smyslu nadstavbou ke strukturovanému.

Prvně si vysvětlíme rozdíl mezi **výrazem** a **příkazem**, a co je to **operátor**. Základním rozdílem mezi výraz a příkazem je tedy to, že výraz je vyhodnocován, a příkaz vykonáván. Operátory znáte běžně z matematiky, jedná se například o plus, mínus, krát či děleno. Operátory se ovšem ještě podle svojí funkce dají dále dělit na:

1. **Aritmetické -** To jsou operátory jako $+, -, \cdot, /$
2. **Logické -** Tyhle operátory určují logický vztah mezi levou a pravou stranou, například AND, OR, NOT, a jejich kombinace.
3. **Relační -** Tyhle operátory určují vztah mezi hodnotami, například $<, \leq, =, >, \geq$

S výrazy, příkazy a operátory jsme nyní schopni popsat spoustu problémů. Stále zde ale máme jisté techniky a principy, které popisování problému razantně usnadňují.

Tabulka 2.1: Hlavní rysy strukturovaného programování

Sekvenční zpracování instrukcí	Instrukce se vykonávají v pořadí tak, jak jsou zapsané od zhora dolů
Funkce a procedury	Blok kódu, pomocí kterého můžeme jednou instrukcí spustit vícero vnořených příkazů
Větvení programu	Rozdělení běhu programu podle logické podmínky
Cykly	Pro opakované vykonávání instrukcí

2.5 Datové struktury

Když se podíváme na nadpis téhle kapitoly, možná vás již napadne, že datové struktury budou něco, co nám pomůže nějakým způsobem strukturovat data. A divte se nebo ne, měli byste pravdu. Stejně jako **datové typy** říkaly, jak hodnoty ukládat v paměti, **datové struktury** nám zase říkají, jak ukládat vícero hodnot v paměti.

Tou nejprimitivnější datovou strukturou je **pole**, které najdete prakticky v každém programovacím jazyce. Pole je velmi dobré znát, jelikož díky němu dokážete vytvořit další datové struktury[1].

[1] Ono je to trochu složitější - lze s nimi tyto struktury vytvořit, ale né vždy budou moci konkurovat jejich nativním implementacím.

Pole má jednu zásadní nevýhodu, a to je jeho fixní délka. Tento problém řeší datová struktura **list** (nebo také známo jako **seznam**), jejíž délku lze libovolně rozšiřovat či zmenšovat - její nevýhodou je však větší (a tím pádem i časově náročnější) režie s pamětí[2].

Další obecnou datovou strukturou je **asociativní pole** (též známo jako **mapa** nebo **slovník**). Tahle struktura dále staví na konceptu *pole* a rozšiřuje možnosti, jak přistupovat k prvkům. K polím přistupujeme pomocí celočíselných indexů, v asociativním poli ale můžeme používat místo celočíselného indexu takzvaný *klíč*.

Příkladem může být např. jazykový slovník, kde klíčem je cizí slovíčko, a hodnotou je český překlad. Nebo třeba telefonní seznam, kde klíčem je jméno člověka, a hodnotou je telefonní seznam[3].

2.6 Výjimky a chybové stavy

Bohužel nastávají situace, kdy naše programy mohou *zpanikařit*. Představme si program, který se chová jako jednoduchá kalkulačka s reálnými čísly a klasickými aritmetickými operacemi. No a teď si představme, že do ní nějaký matlák[4] nacvaká $\frac{1}{0}$. Nulou v reálných číslech dělit neumíme, takže, co teď? Zpravidla by program takzvaně *vyhodil* či *vyvolal* **výjimku**.

Výjimkou tak vyvolá například již výše zmíněné dělení nulou, nebo třeba čtení neexistujícího souboru. Výjimky jsou takto pojmenované právě kvůli tomu, že se jedná o *výjimečné stavy* programu, a programovací jazyky nám dávají možnost tyto stavy ošetřit a program *zotavit*. Nad výjimkou ovšem stojí ještě **chybový stav**.

Chybou může být například špatné přidělování paměti či chyba způsobená nižším rozhraním. Takovým typickým příkladem chybového stavu ze života je *modrá obrazovka smrti*. Z chyb se program kvůli jejich vážnosti nezotavuje - prakticky by to ani nebylo možné.

[2] Ono ani tohle není úplně pravda - rozhodně zabírá více místa v paměti, ale pak už záleží na implemetaci (např. zřetězení prvků pomocí odkazů na další prvek má režii minimální)

[3] Nebo naopak. Zpravidla jsou asociativní pole jednoznačná, takže lze vytvářet inverze a z hodnot udělat klíče a naopak

[4] Čtěte jako matlabák

3 Úvod do objektového programování

Objektově orientované programování (dále už jenom OOP) je jedno z nejrozšířenějších programovacích paradigmat. Nachází velké rozšíření napříč programovacími jazyky a je de-facto jedním z nejpoužívanějších paradigmat. Jeho výhodou je rozhodně jednoduchost návrhu řešení problému - OOP se totiž snaží simulovat reálný svět tím, že problém popíše pomocí různých entit, které mezi sebou navzájem interagují a dohromady řeší problém. Je to velmi podobné například manufaktuře, kde každý pracovník dělá jenom konkrétní činnost, jejíž výsledek poté předává dál - a na konci pak čeká již hotový výrobek.

Ačkoliv je OOP velmi jednoduchý na pochopení, tak je také velmi jednoduchý na pokažení. OOP se dokáže velmi zvrtnout, velmi rychle. Jsou také případy, kdy OOP nepřináší žádnou výhodu a jiné paradigma by možná bylo užitečnější - to už pak záleží na zdrojích, které nám poskytuje konkrétní programovací jazyk.

Nenechte se tímto odradit, jenom jsem chtěl upozornit na to, že OOP není blbuvzdorné a může vám v případě špatného použití spíše zavařit než-li pomoci.

3.1 Objekty

Základním stavebním blokem objektově orientovaného programování je, jak už název napovídá, **objekt**. Jako objekt se v objektově orientovaném programování označuje ledacos, a je to pojem spíše nápomocný k vysvětlování dalších pojmů. My budeme chápat jako objekt strukturu, která si uchovává dvě podstatné věci:

- **Data** - Informace a hodnoty, se kterými může objekt nějak pracovat
- a **Operace** - Věci, které dokáže provést

Data a operace nejsou ovšem není zrovna správné pojmenování těchto věcí - častěji se setkáte s jejich odbornými termíny, což jsou **atributy** (data) a **metody** (operace). Každý z tento pojmů postupně probereme.

3.1.1 Atributy

Jako atributy označujeme proměnné nebo konstanty, ve kterých si objekt uchovává informace a hodnoty, které dále ve svém kódu využívá. Tyto atributy jsou nejčastěji používány buďto pro sdílení informací napříč jednotlivými metodami, nebo pro nastavování nějakého obecnějšího chování.

Představme si například objekt **obdélník**. Každý obdélník má **délku** a **šířku**. Délka a šířka jsou tedy atributy objektu obdélník. Tyto atributy můžeme dále používat pro výpočet obvodu, obsahu či úhlopříček...

3.1.2 Metody

Metody jsou operace, které může nějaký konkrétní objekt provádět. Jedná se konkrétně buďto o funkce nebo procedury, které má objekt k dispozici.

Metody jsou hnacím motorem daného objektu - slouží k dělání výpočtů, manipulaci s dalšími objekty, vytváření dalších objektů, zkrátka objektu dají nějakou funkcionalitu.

3.2 Vytváření objektů a přístup k objektům

Zatím jsme si popsali objekt jako takový, ale neřekli jsme, odkud se objekty berou. Aby programovací jazyk věděl, jakým způsobem má nějaký objekt vytvořit, potřebuje pro něj nějaký **předpis**. Tomuto předpisu se říká **třída**.

3.2.1 Třída objektu

Třída objektu předepisuje, jaké vlastnosti a metody bude daný objekt mít. Vzpomeňte si na naší definici objektu - je to věc, která si uchovává v paměti atributy a metody. Třída nedělá nic jiného, než že říká, co tyhle vlastnosti a metody jsou konkrétně zač.

S předpisem v ruce může programovací jazyk začít tvořit objekty - kde má ale začít? K tomu lze využít buďto již zabudovanou, nebo námi vytvořenou speciální metodu, které se říká **konstruktor**.

3.2.2 Konstruktor a destruktor

Konstruktor je metoda, kterou má každá třída, a která říká programovacímu jazyku, jak má vytvářet nové objekty. Každá třída má svůj tzv. *implicitní konstruktor* - to je takový, který se použije, pokud tam nemáme svůj vlastní (respektive nazvaný *explicitní konstruktor*).

Konstruktory se používají k naplnění objektu daty, které můžeme pomocí argumentů předat z vnějšího programu. Vždy je výsledkem konstruktoru nová instance objektu - zpravidla lze vytváření objektu pomocí konstruktory přerušit akorát vyvoláním výjimky. Existují principy, jak vytvářet objekty a případně zamezit jejich vytvoření, ale to je mimo rozsah této příručky. Analogicky k metodě, která vytváří objekt, také existuje metoda, která objekt níčí (Odborníci čtěte jako *uvolňuje z paměti*). Té se říká **destruktor**.

3.2.3 Modifikátory přístupu

Jak už název napovídá, modifikátory přístupu *modifikují přístup* - otázkou je, přístup k čemu modifikují? Odpovědí je tak nějak ke všemu.

Tahle definice se může zdát trochu vágní - vysvětlím tedy na příkladech. Nejdříve ale konkrétní modifikátory přístupu - nejčastěji se setkáte s dvojicí **public** a **private**. **Public** strukturu prezentuje celému zbytku programu, kde jakákoliv další struktura k němu má přístup, naopak **private** strukturu schová.

Poznámka: Pod pojmem "struktura" si zde představte například třídu, atribut či metodu. Všechny tyto zpravidla bývají modifikovatelné z hlediska přístupu.

Příkladem budiž privátní metoda nějaké třídy - takovou metodu nelze zavolat z vnějšku, ale lze ji používat akorát v dané třídě. Privátní třídy také existují, ale to je trochu absurdní a málo používaná modifikace přístupu.

3.3 Interakce mezi objekty

Již jsme si pověděli něco o objektech a odkud se berou - pojďme se nyní podívat, jak můžeme používat více druhů objektů k vytvoření našeho programu. Nejdříve si vysvětlíme dva pojmy, se kterými se v lekcích OOP setkáte.

3.3.1 Abstrakce a zapouzdření

Za tuhle definici by mě "kolegové" zabili, ale říká v podstatě hlavní myšlenku vytváření abstrakcí - a to je zjednodušování (či zobecňování) našeho programování. V OOP to znamená, že vytváříme prostředky, jejíž operace poté dohromady používáme k vytvoření složitého algoritmu pomocí "jednoduchých" příkazů.

Do jaké míry jsme proces zjednodušili označujeme jako **úroveň abstrakce**. S tímto termínem se často setkáte v povídání o programovacích jazycích, protože jsou jazyky s

1. **nízkou úrovní abstrakce**, které jsou pro lidi nečitelné, ale nevyžadují složitý překlad pro počítač
2. nebo **vysokou úrovní abstrakce**, které jsou blíže lidskému jazyku, ale vyžadují složitější překlad/interpretaci.

S abstrakcí úzce souvisí následující pojem, a to je **zapouzdření**. Jak již víme, abstrakce je proces zjednodušování složitých operací. Stejně jako v reálném světě platí, že jednoduchost je v nepřímé úměrnosti s počtem možností - čím méně toho víme, tím se to jeví jednodušší[1], a v programování se můžeme setkat s podobnou paralelou - je lepší, když máme jeden konkrétní způsob, jak něco udělat, než 5 různých, lehce odlišných způsobů.

[1] Této heuristice se říká Hickův zákon

Zapouzdřením uděláme kolem našeho objektu ”pouzdro”, které schová vnitřnosti objektu a nechá odhalené akorát to, co chceme, aby používaly věci z vnějšího programu. Často se to místo pouzdra přirovnává k ”černé skřínce”, do které nevidíme, ale víme, co dělá.

Můžeme tedy říct, že zapouzdření kontroluje tzv. ”vstupní body”, kterými dovolíme vnějšímu programu komunikovat s naším objektem. Nejčastěji se tohle používá k zamezení nebezpečnému manipulování s pomocnými proměnnými nebo konfiguračními atributy.

3.3.2 Skládání objektů

Pojďme trochu provázat naše objekty - tomu vysocí páni (a paní) říkají **skládání**[2] **objektů**. Není to nic jiného než využívání metod cizího objektu v metodách objektu.

3.4 Statická deklarace

Zatím vše, co jsme si ukázali, tak se nějak či onak vázalo na *objekt*, respektive *instanci třídy*, co kdybychom ale chtěli vytvořit konstantu, kterou půjde využít napříč všemi instancemi třídy? Naivní cesta by byla manuální přiřazení v konstruktoru, ovšem OOP jazyky disponují nástrojem, kterému se říká *statická deklarace*.

Teď teda co to konkrétně znamená - představme si *obdélník*. Jak velký obdélník jste si představili? Kolik si jich můžeme představit? Obdélníků je nekonečně mnoho, protože vždy dokážeme vytvořit nový s jinými rozměry (vnitřním stavem). Každý obdélník má ale jednu věc stejnou, a to je způsob, kterým vypočteme jeho obsah. Výpočet obsahu bychom tedy mohli nazvat *statickou metodou třídy obdélník*.

3.5 Dědičnost

Dědičnost je principem, který v jistém slova smyslu rozšiřuje pojem *skládání objektů* až na úroveň tříd. Představme si, že máme třídu, a potřebujeme ještě jednu, která je v podstatě stejná, akorát se liší v implementaci jedné metody. Trochu blbý způsob by byl třídu zkopírovat. Mnohem lepší by bylo novou třídu *odvodit* a následně *překrýt* (přepsat implementaci) metodu.

Odvozené třídy mají všechny atributy a metody z třídy, ze které dědí. Rozdíl je v tom, že odvozená třída má možnost původní metody překrýt, což znamená přepsat vlastním kódem.

Dědičnost se často používá ve spojení s abstrakcí k vytvoření takzvané *hiearchie tříd* - než abych chodil kolem horké kaše, ukažme si takovou *hiearchii* na příkladu.

Uvažujme o lidech - (většinou) každý člověk umí chodit, psát, mluvit, a do jisté míry i počítat. Né každý ale umí naprogramovat bankovní systém. Všichni lidé mají nějaké základní vlastnosti, ale jenom menší hrst má ještě navíc nějaké další konkrétní vlastnosti. Než abychom museli pro programátora, popeláře, studenta či politika

[2] Pojem *skládání* vychází z matematiky, konkrétně ze složených funkcí.

zvlášť implementovat to, že umí mluvit, tak tyto společné vlastnosti přesuneme do vyšší třídy - třídy člověk, ze které odvodíme jednotlivá povolání.

3.6 Polymorfismus

Polymorfismus je Achillova pata testů z teorie OOP, přičemž to není vůbec složitý koncept - jenom jméno je zbytečně složité. Pojďme si tohle slovo rozebrat:

- Předpona **poly** znamená *více* nebo *mnoho*.
- Kořen **morf** znamená *tvar* - vzpomeňte si třeba na seriál *Ben 10* nebo podobné: když někdo *morfuje*, tak zpravidla mění tvar.
- Koncovka **-ismus** zpravidla označuje nějakou vlastnost

Tahle interpretace by nám pověděla, že **polymorfismus** bychom mohli nahradit spojením slov **mnoho, tvar a vlastnost**, jinak řečeno, **mnohotvárnost**.

Opět né úplně přesná definice, ale v podstatě říká, že objekt by neměl být nutně vázaný na jeden typ dat, pokud to není nezbytně nutné. Takový objekt *nůž* by neměl umět krájet akorát *jablka*, ale i *hrušky* nebo *chleba*.

4 Závěr

Práce se snažila ”polopatě” vysvětlit, jakým způsobem přistupovat k problémům algoritmizace a objasnit základy strukturovaného a objektově orientovaného programování.

Literatura

[1] FINĚK, Václav. *Matematika 1 a 2* [online]. Liberec, 2022 [cit. 2022-04-12]. Dostupné na WWW: https://kmd.fp.tul.cz/images/stories/vyuka/finek-matematika1/Matematika_11.pdf

[2] KRÁLOVCOVÁ, Jiřina. *Materiály k přednášce Algoritmizace a programování 2* [online]. Liberec, 2022 [cit. 2022-05-16].

A Slovník pojmů

Programování	Proces vytvoření programu, který řeší nějaký problém.
Algoritmus	Konečný sled instrukcí, který vede k řešení problému, který má algoritmus řešit.
Algoritmizace	Proces vytváření algoritmu.
Asymptotická složitost	Funkce, která nám říká, jak rychle roste nějaká veličina společně s velikostí vstupních dat. Platí, že čím pomaleji roste, tím je algoritmus efektivnější.
Datový typ	Určení, jakým způsobem budou data uložena v paměti, a co s takovými daty můžeme provádět.
Přetypování	Proces převodu mezi datovými typy, pokud je takový převod možný.
Výraz	Sekvence konstant a proměnných, které jsou spojeny operátory a které jsou vyhodnocovány.
Příkaz	Volání procedury nebo funkce, které jsou vykonávány.
Operátor	Určuje vztah mezi věcí nalevo od něj a napravo od něj.
Procedura	Spustitelný blok kódu, který obsahuje sled instrukcí.
Funkce	Procedura, která na konci (nebo někdy během svého běhu) vrátí nějaký výsledek.
Podmíněné větvení	Struktura, která podle pravdivosti výrazu rozhodne, který blok kódu má spustit.
Cyklus	Struktura, která slouží k opakování bloků kódu.
Tělo cyklu	Blok kódu, který bude smyčkou opakovaně vykonáván.
Iterace	Jedno vykonání kódu v těle cyklu.
Pole	Datová struktura, která má fixní počet prvků. Tyto prvky jsou řazeny za sebou a lze k nim přistupovat pomocí indexu.
Index	Pořadové číslo prvku v poli.
Asociativní pole	Datová struktura, která je složená z dvojic, klíč a hodnota. K hodnotě se přistupuje pomocí klíče.
Výjimka	Událost, která nastane v případě, kdy program nemůže nějakou operaci/instrukci provést.
Chyba	Událost, která nastane při selhání prostředků počítače za běhu programu.
Objekt	Základní jednotka objektově orientovaného programování, která si uchovává **atributy** (data) a **metody** (operace, které dokáže provést).

Atribut	Informace (ať už proměnná či konstantí), kterou si objekt uchovává v paměti.
Metoda	Pojmenovaný blok kódu v danému objektu, který může objekt volat.
Třída	Předepis objektu, jeho atributů a metod, společně s jejich datovými typy a modifikátory přístupu.
Konstruktor	Speciální metoda třídy, která se volá při vytváření objektu.
Instance třídy	Objekt, který byl vytvořen pomocí konstruktoru nějaké třídy.
Modifikátory přístupu	Sada klíčových slov, které říkají, jakým způsobem lze ke struktuře přistupovat z vnějšku.
Abstrakce	Proces, který složité úkony schová za jednoduché rozhraní.
Zapouzdření	Koncept, kdy pomocí modifikátorů přístupu kontrolujeme, jak a co bude přístupné z vnějšího programu.
Skládání objektů	Proces, ve kterém v metodě jednoho objektu používáme prostředky cizího objektu.
Statická deklarace	Vytvoření (deklarace) struktury, jejíž hodnota není závislá na vnitřním stavu instance objektu.
Dědičnost	Koncept dovolující vytvářet nové objekty a struktury na základě již dříve definovaných objektů/struktur.
Polymorfismus	Vlastnost objektů, která říká, že nezáleží na typu dat, jelikož je objekt dokáže obsloužit.