Fakulta informatiky a statistiky Vysoké školy ekonomické v Praze, Katedra informačních technologií

Bakalářská práce

Alternativní výukové materiály pro vstupní kurzy programování

Vedoucí práce: Ing. Rudolf Pecinovský, CSc.
Oponent: Ing. Jarmila Pavlíčková

Autor: Jakub Závěrka
12.5.2010

PROHLÁŠENÍ

Prohlašuji, že jsem svoji bakalářskou práci na téma „Alternativní výukové materiály pro vstupní kurzy programování“ vypracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem při zpracování čerpal.

V Praze dne …… ……

Jakub Závěrka

PODĚKOVÁNÍ

Tímto bych chtěl poděkovat ing. Pecinovskému za vstřícný postoj ke zpracování práce, za poskytnuté rady a počítačové programy, za zapůjčenou literaturu a především za čas, který mi věnoval při konzultacích této práce.

Jakub Závěrka

ABSTRAKT

Tato bakalářská práce vytváří alternativní výukové materiály pro vstupní kurzy programování ve studijním programu Informatika na Fakultě informatiky a statistiky Vysoké školy ekonomické v Praze. Konkrétně se zabývá kurzy *4IT101 Základy programování* a *4IT115 Základy softwarového inženýrství*, s důrazem kladeným na první z těchto předmětů.

Práce analyzuje způsob, jakým jsou studenti seznamování s konceptem chyb v programu a jejích vyhledáváním a opravováním. Vyhodnocuje pokrytí tohoto tématu v literatuře (zejména v publikacích uvedených v sylabu předmětu 4IT101) a porovnává situaci mezi studenty těchto kurzů.

Hlavní část práce tvoří zpracování materiálů v podobě textu a krátkých instruktážních videí na téma ladění programu. Studenti se pomocí této práce dozvědí, jak chyby v jejich programech vznikají, jak se projevují, jak se odhalují a jak se opravují. Vedlejším produktem této práce je také revize konfiguračních souborů pro program BlueJ. V těchto souborech došlo k úpravám zpráv, které program BlueJ vrací v případě chyby. Tyto zprávy byly přepsány a doplněny tak, aby byly pro začátečníky srozumitelné.

Praktická část práce je zpřístupněná jako HTML soubor na Internetu.

Klíčová slova:

Java, ladění programů, výuka programování, 4IT101

ABSTRACT

This bachelor thesis creates alternative educational materials for beginner programming courses in the Informatics curriculum at the Faculty of Informatics and Statistics, University of Economics, Prague. Specifically, these courses comprise of *4IT101 Introduction to programming* and *4IT115 Fundamentals of Software Engineering*, with emphasis on the former one.

In specific, this thesis analyses the way students are introduced to the conception of software bugs and how these bugs are searched for and corrected. It evaluates the coverage of these problems in literature (especially in literature for the course 4IT101) and compares situation among student of the course.

The main part of this thesis consists of working up materials in form of text and short videos oriented towards program debugging. This thesis will help the students to learn how bugs originate, how they demonstrate themselves, how they can be detected and ultimately, corrected. As a side-effect of this thesis, the author revised configuration files for the application BlueJ. The messages that are returned to the user by the application in case of an error or exception were modified in these files. These messages were rewritten or expanded to be more comprehensible for a beginner programmer.

These materials are accessible as an HTML page on the Internet.

Keywords:

Java, debugging, programming education, University of Economics Prague, 4IT101

OBSAH

1 ÚVOD

1.1 VÝBĚR TÉMATU

Přesná podoba tématu této bakalářské práce se vyvíjela během několika konzultací autora s vedoucím práce. Původní nápad byl zprostředkovat studentům vstupních kurzů programování jednoduchou a přehlednou příručku k programovacímu jazyku Java, která by se nezabývala pokročilými konstrukcemi nebo složitějšími principy, ale pomohla by studentům pochopit způsoby, kterými je program stavěn. Nicméně po dohodě s ing. Pecinovským, který správně podotkl, že toto téma bylo již mnohokrát zpracováno jinými autory a přínos práce by tedy byl marginální, došlo ke změně tématu.

Cílem této práce je stále pomoci studentům vstupních kurzů, aby se v problematice objektově orientovaného programování náležitě zorientovali. Práce se však nezabývá programováním jako celkem, nýbrž se zaměřuje na hledání a opravování chyb, které studenti těchto kurzů nevyhnutelně ve svých programech udělají. Smyslem této práce je vysvětlit studentům úskalí softwarových chyb, měla by jim pomoci pochopit, jak chyby vznikají, jak se vyhledávají a jak je ve svých programech opravit. Tato činnost vyhledávání příčiny chyby a její oprava je v programátorských kruzích nazývána ladění (anglicky debugging).

Ve zpracování autor zejména využil tříleté zkušenosti se vstupními kurzy a jejich osnovami. Autor předně kurzy sám absolvoval, co je ale pro zpracování podstatné, nabízel doučování pro studenty těchto kurzů i v dalších semestrech. Detailně se tak seznámil se všemi způsoby, jak studenti mohou výuku programování pojmout, a nabral zkušenosti při výuce programování. Především ale zjistil, co těmto studentům dělá při programování největší problém a co je pro ně nejobtížnější pochopit.

Jedna ze zkušeností autora je taková, že po absolvování druhého vstupního kurzu (4IT115) se studenti dají rozdělit pouze na dvě skupiny: jedni, které programování zaujalo, neboť pochopili jeho principy, a druzí, kteří si k programování vyvinuli velice negativní vztah, protože jeho principy nepochopili. Hlavní myšlenkou této práce je zamezit potencionálním adeptům na členství v druhé skupině, aby se členy skutečně stali, ale naopak je přivést do první skupiny i v případě, že při vytváření jejich programů nejde vždy vše tak, jak si představují.

V konečném důsledku pro absolventy studijního programu Informatika členství v první skupině znamená lepší uplatnění na poli informačních technologií, neboť tito absolventi budou schopni pracovat i na pozicích spojených s vývojem softwaru; na pozicích, kterým se budou členové druhé skupiny spíše vyhýbat. Pro fakultu informačních technologií na VŠE to bude znamenat zlepšení již tak vynikající možnosti uplatnění absolventů v oboru a zvýšení konkurenceschopnosti těchto absolventů v porovnání s absolventy ostatních informatických oborů na jiných vysokých školách.

1.2 PRINCIPY ZPRACOVÁNÍ

Jeden z předpokladů této práce je analýza pokrytí nastíněné problematiky v literatuře. Autor se primárně zaměřil na výukové knihy uvedené v sylabu kurzu 4IT101 (publikace [1; 2; 3; 4]), ale do analýzy zahrnul i další knihy od českých i světových autorů. Z těchto knih bych vyzdvihl novou výukovou publikaci *OOP – Naučte se myslet a programovat objektově* [5] od ing. Pecinovského, která studentům vysvětluje látku ve formě otázek a odpovědí. Výše zmíněné knihy a tato další jsou v současnosti nejpřístupnějšími publikacemi pro zájemce o programování.

Na základě této analýzy bude vypracován alternativní výukový materiál pro vstupní kurzy programování. Tento materiál bude primárně zaměřen na ty studenti, jež mají při tvorbě svých programů problémy, které nejsou v základní literatuře pokryty. Nemá za cíl suplovat literaturu vstupních kurzů, ale naopak ji bude vhodně doplňovat o problematiku vyhledávání a opravování chyb.

1.3 ANALÝZA PUBLIKACÍ

V rámci této práce proběhla analýza několika knih, které o sobě tvrdí, že vyučují programování v Javě [1; 2; 3; 4; 5; 6; 7; 8; 9; 10], ale ani jedna nepokrývá problematiku chyb a ladění zcela dostatečně. Učí programování, ale neuvědomují si přitom, že programování není pouze o tom psát syntakticky čistý kód v daném jazyku. Knihy čtenáře seznamují s téměř všemi konstrukcemi Javy a v některých případech probírají i některé nejpoužívanější funkce standardní knihovny, ale toto nestačí. Nelze předpokládat, že začátečníci nebudou ve svých programech dělat chyby, jakkoli jsou ukázkové programy triviální, a nemohou si tedy dovolit kapitolu o ladění vypustit.

Pro začátečníky je testování, ladění a oprava chyb nevyhnutelná činnost, s níž by měli být seznámeni co nejdříve. S pokročilými konstrukcemi a funkcemi standardní knihovny je lepší seznamovat až pokročilé studenty. Bohužel, téměř žádná kniha z výše uvedeného seznamu se ladění nevěnuje na úrovni, která by byla pro začátečníky alespoň přijatelná (zato ale spousta knih popisuje pokročilé knihovny nebo dokonce problematiku vláken).

Následuje analýza publikací, které jsou uvedeny v přehledu literatury ke kurzu 4IT101 (kniha *Podniková informatika* od L. Gály, P. Tomana a J. Poura byla vynechaná, protože se programování nevěnuje). Některé další knihy si také zaslouží analýzu jejich přístupu k ladění – ty jsou uvedeny později.

1.3.1 ÚVOD DO JAVY [1]

Kniha Úvod do Javy popisuje základní principy práce se syntaxí Javy. Osvětluje studentům také základy práce s objekty a prezentuje některé nejzajímavější knihovny, které jsou součástí JRE. Tato publikace je pro kurz 4IT101 zařazena mezi základní literaturu, lze se tedy domnívat, že většina absolventů s ní přijde do styku (i mimo jiné proto, že je veřejně přístupná na serveru java.vse.cz).

Bohužel kniha se nijak nevěnuje konceptu softwarové chyby ani ladění obecně. Popisuje strukturu výjimek a práci s nimi, ale problematiku vykládá spíše jako další objektovou konstrukci v Javě, než aby se zabývala příčinami výjimek. Popisuje aspoň některé nejčastější výjimky a přikládá k nim uspokojující popis.

V dalších kapitolách se kniha pouze okrajově zmiňuje o některých dílčích problémech. Například se zmiňuje o chybách při vstupu a výstupu (I/O), ale opět s přístupem, že tyto výjimky mohou nastat a musí se odchytit, protože to překladač vyžaduje.

1.3.2 OBJECT-FIRST WITH JAVA [2]

Touto knihou se nechala inspirovat skripta od manželů Pavlíčkových [1]. Některé projekty řešené v základním kurzu (např. tvary, kalkulačka nebo textová agentura) jsou původně řešené zde. Kniha využívá BlueJ jako ukázkové prostředí (M. Kölling je jedním ze spoluautorů programu BlueJ).

Problematiku debuggingu kniha nastíní jako činnost, která následuje po testování a vyjmenuje nejdůležitější metody debuggingu, včetně těch základních. Kniha se věnuje především testováním (a to hlavně jednotkovými testy), ladění je zde pokryto již v nižší míře. Ale i tak je možno uvést, že dostatečně – jsou zde popsány manuální průchody kódem, ladící výpisy a debugger, a to poměrně detailně. Debugger popisuje jako nástroj, který lze využít ke sledování kódu, během kterého uživatel může chybu odhalit. Kniha dokonce poskytuje na přiloženém CD projekt, který obsahuje několik chyb. Uživatel tedy může vyzkoušet techniky ladění ihned v praxi.

Bohužel, kniha neobsahuje ani malý seznam nejběžnějších chyb, na které může uživatel narazit. Předpokladem úspěšného a rychlého ladění je alespoň minimální znalost možných příčin – a tu tato kniha neposkytuje.

1.3.3 MYSLÍME OBJEKTOVĚ V JAZYKU JAVA [3]

V této knize je celé problematice ladění věnována pouze jedna kapitola (3.7 – Ladění) o zhruba pěti stránkách. Čtenář se zde může seznámit se základním dělením chyb na syntaktické, běhové a logické. Kniha čtenáře v této kapitole ale pouze seznámí s těmito chybami a jejich základními příznaky, nepoukazuje na možné příčiny a nenavrhuje žádné řešení.

Dále se čtenář dozví, co to debugger a jak se s ním pracuje. Bohužel již zde není dostatečně jasně vysvětleno, k čemu je debugger dobrý. V dalších kapitolách kniha nastiňuje některé problémy (např. nekonečný cyklus nebo hodnotové limity primitivních typů), ale pouze je zmíní jako problém a dál je neanalyzuje.

Obecně se kniha věnuje především syntaxi a správnému objektovému návrhu. Chyby v programech neřeší systematicky, pouze na ně upozorňuje, když se naskytnou. Ladění vynechává téměř úplně.

1.3.4 HEAD FIRST JAVA [4]

Poměrně unikátní kniha, která podává programování v Javě vtipnou a zábavnou formou. Využívá nápadité abstrakce problémů, což studentům usnadňuje pochopení látky. V některých případech se ale abstrakcí od podstaty problému odchýlí více, než je nutné.

Chybám a ladění se kniha téměř nevěnuje. Zmiňuje některé možnosti (výjimky), ale uvede je pouze jako jednu z konstrukcí jazyka. I když se zařazuje jako kniha pro začátečníky, později se čtenář seznámí s některými funkcemi standardní knihovny, které lze přímo nazvat pokročilé – např. vstup / výstup nebo GUI z knihovny Swing.

1.3.5 OSTATNÍ KNIHY

Co se týče dalších knih, má smysl se zmínit pouze o těch, které k problematice ladění přistupují alespoň trochu svědomitě nebo o těch, u kterých je rozumná možnost, že by se s nimi student vstupních kurzů na VŠE mohl setkat.

1.3.5.1 OOP – Naučte se myslet a programovat objektově [5]

Tato poměrně nová kniha (2010) objektového programování je primárně určena pro střední školy. Materiál kniha podává ve formě otázek a odpovědí. OOP kniha vykládá metodikou design-pattern-first[1]. Nicméně, kniha klade veliký důraz na správný návrh a čtenáře detailně seznamuje s principy OOP, než se dostane k tvorbě kódu. Čtenář tak může mít pocit, že je programování příliš složité, když se před tím musí prokousat dvěma sty stranami teorie.

Ladění programů s publikace téměř nezmiňuje. Některé časté chyby řeší, když se k nim čtenář dostane, ale jinak zde není uveden žádný jednotný pohled na vyhledávání a odhalování chyb.

1.3.5.2 Big Java [6]

Tato poměrně obsáhlá kniha popisuje jazyk Java do nejmenších podrobností. Kniha seznamuje čtenáře se spoustou pokročilých technik a konstrukcí, zároveň ale v každé sekci může čtenář narazit na nejčastější chyby a jejich opravu. Kniha zároveň pro složitější látku poskytuje doporučené „how-to“ postupy.

Ladění kniha pojímá jako činnost následující po testování, nastíní programovací výpisy, lehce se zmíní o logování a popíše debugger. Zmiňuje také aserce. Rovněž popisuje čtenářům doporučené postupy při ladění, a to poměrně úspěšně.

Na knihu o dvanácti set stránkách je ladění pokryté v tom nezákladnějším dostatečném množství. Čtenáře seznámí s nejčastějšími chybami, laděním a doporučenými postupy, tuto problematiku ale dál nerozvádí a je na čtenáři, aby si sám vyhledal o problematice další zdroje.

[1] Design-pattern-first = výukový přístup založen na tom, že studenti by se nejprve měli naučit zaběhnuté a osvědčené objektové programátorské postupy (tzv. návrhové vzory – design patterns), než se začnou věnovat plně programování nebo se seznámí se složitějšími strukturami.

1.3.5.3 Java 6 – výukový kurz [7]

Tato kniha jako jediná uvádí běžné problémy a jejich řešení. Uvádí, co může způsobit při překladu jaké chyby. Zmiňuje rovněž chyby při běhu, a jak se projeví. Bohužel, kniha používá zabudovaný kompilátor `javac.exe` a řeší jeho zprávy (kompilovat si třídy přes příkazový řádek je dnes již překonaný postup).

1.3.5.4 Java – začínáme programovat [8]

Tato kniha má samostatnou podkapitolu o ladění. Popisuje v ní postup ladění pomocí ladících výpisů a debuggeru. Pro ladící výpisy však používá výpis System.err, místo System.out. Navíc popisuje zabudovaný javovský debugger, který se ovládá přes příkazovou řádku, a jehož ovládání je tím pádem extrémně nepohodlné. Tyto překonané postupy pravděpodobně autor uvádí proto, že kniha pochází z roku 2002.

Dále kniha zmiňuje pokročilé ladící nástroje, jako je Apache Log4J nebo Java Logging API. Pro základní ladění neposkytují tyto komplexní nástroje takovou přidanou hodnotu, ale přesto se jim autor věnuje poměrně detailně.

1.4 SHRNUTÍ

Ani v jedné z analyzovaných publikací není problematika debuggingu dostatečně řešena. Je až s podivem, že učebnice, které jsou určeny i těm největším začátečníkům, se nevěnují tak zásadní činnosti, kterou ladění při vytváření programů bezesporu je. Pokud studenti narazí na problém, se kterým si neví rady (a v učebnici není popsán), je pro ně velice obtížné tento problém řešit, pokud ani neví, jak ho správně uchopit.

Samozřejmě, že existují knihy věnované debuggingu (jako např. kniha *Debugging* od autora Davida J. Aganse nebo *The Developer's Guide to Debugging* od autorů Thorsten Grötker, Ulrich Holtmann, Holger Keding a Markus Wloka), nelze ale předpokládat, že by začátečníci tyto knihy vyhledávali sami. A ačkoliv je v sylabu kurzu 4IT101 uvedeno „testování a ladění aplikací“, z autorovy zkušenosti (a zkušenosti dalších studentů) se tomuto aspektu vývoje softwaru vyučující příliš ve cvičeních nevěnují (a ani není žádná kniha o ladění uvedena mezi doporučenou literaturou). Studenti tedy nemají téměř žádnou možnost se naučit ladit. Nevyhnutelným důsledkem je fakt, že zejména pomalejší studenti, kteří se s chybami ve svých programech setkávají nejčastěji, začínají být frustrovaní a vyvinou si vůči programování psychologickou bariéru. Tito studenti se pak již těžko této bariéry zbavují a zcela zbytečně se připravují o další možnosti uplatnění (jak bylo popsáno na konci podkapitoly 1.1).

Tato práce se pokouší toto prázdné místo alespoň částečně vyplnit. Nabízí praktický manuál na nejčastější chyby, se kterými se studenti vstupních kurzů programování na VŠE setkají. Když studenti pochopí principy těchto chyb a ukáže se jim i modelové řešení, budou schopni si tyto chyby v budoucnu opravit sami. Získají tak důležitou zkušenost s laděním, která jim pomůže nalézt a opravit chyby i v dalších kurzech, kde se setkají

s jinými chybami a nebudou mít při ruce příručku, která by jim řekla, co tyto chyby znamenají a jak je opravit.

1.4.1 FORMA PRÁCE

Práce je zpracována ve formě příručky, která bude přístupná v on-line formě na serveru java.vse.cz (nebo jinde, kam mají studenti volný přístup). V příručce budou nastíněny nejzávažnější problémy, se kterými se studenti setkávají, a popis jejich řešení. Pro analýzu některých složitějších problému se využívá animačního programu Wink (http://www.debugmode.com/wink/), kterým bude zachycena obrazovka s analýzou a řešením problému. Studenti tak uvidí postup činnosti v reálném čase. Tato veřejná část nebude používat formální styl, ale spíše neformální styl, aby se problematika více přiblížila čtenářům – začátečníkům.

2 JAK HLEDAT CHYBY V PROGRAMECH

2.1 ÚVOD

2.1.1 JAK CHYBY VZNIKAJÍ

Každý programátor je vždy jen člověk. Člověk je z podstaty věci nedokonalý. Cokoliv, co člověk dělá (stavařinu, vaření, učitelství, politiku), nedělá bez chyb. Stavitel může špatně navrhnout most, který spadne, kuchař může přesolit polévku. Stejné je to i s programátory – také svoji činnost nedělají perfektně, ale téměř vždy jejich programy budou obsahovat chyby.

Nejčastěji chyby vznikají tak, že programátor na něco prostě zapomene. Zapomene vytvořit objekt, zapomene inicializovat proměnnou, zapomene zavolat klíčovou metodu, atd. Některé chyby vznikají tak, že se programátor uklepne a udělá překlep, který překladač programu potom nezpracuje. Oba dva tyto typy jsou poměrně jednoduché a snadno odhalitelné. Pak jsou tady ještě ale další chyby, které vznikají tak, že programátor něco naprogramuje, ale myslí si, že naprogramoval něco jiného. Jednoduchý příklad: programátor pro třídu `Tvar` naprogramoval metodu `posuňDoleva()`, kde k x-ové souřadnici tvaru přičte 10. Programátor si myslí, že naprogramoval metodu, která posune tvar doleva (navíc se ta metoda tak jmenuje), ale neuvědomil si, že musí od souřadnice odečítat, ne přičítat, když chce tvar posunout doleva. Po zavolání metody `posuňDoleva()` se pak tvar posune doprava.

Této nerovnosti se programátor velice těžko zbavuje a odhalení těchto chyb může být poměrně složité. Je totiž obecně známo, že lidé po sobě chyby nevidí. Proto autoři nějakého důležitého textu, např. knihy, nechávají tento text projít jazykovou korekturou. Cizí osoba má totiž větší šanci odhalit přítomné chyby. A stejný princip platí jak o psaní, tak o programování.

Formálně lze tedy chyby v Javě rozdělit do třech skupin:

1) **Syntaktické chyby** – tyto chyby se projevují tak, že nedovolí programu, aby se vůbec přeložil. Překladač obecně při překladu uživateli napíše, co brání přeložení kódu a v jakém souboru a na jakém řádku byla chyba nalezena.
2) **Běhové chyby** – tyto chyby nebrání překladu kódu, nicméně při spuštění programu nebo při uživatelské akci se program dostane do nestandardní situace, kterou program ohlásí vygenerováním výjimky nebo systémové chyby (instance tříd `Exception` nebo `Error`). Takováto chyba může být programátorem zachycena a zpracována, ale často není zpracování chyby možné. Programové vlákno, ve kterém se takováto chyba stala, je poté ukončeno.
3) **Logické chyby** – nejzávažnější chyby, kterých se programátor může dopustit. Tyto chyby jsou závažné tím, že často nemají žádné bezprostřední symptomy

nebo se projevují pouze nepřímo. Z pohledu překladače ani JVM nedojde k žádné nestandardní situaci a programu je povoleno pokračovat. Podstatou chyby je rozpor mezi tím, co má program dělat, a tím, jak byl naprogramován.

2.1.2 JAK CHYBY NAJÍT

Naštěstí se všechny chyby dříve nebo později projeví tak, že program začne dělat něco jiného, než má dělat nebo od něj čekáme. Pak následuje prohlídka kódu, který je zodpovědný za danou činnost a jeho analýza. Je důležité si uvědomit, co by mohlo být příčinou problému – u hledání chyby je nutné myslet. Například pro příklad uvedený výše by chyba mohla mít dvě příčiny: chybný kód metody `posuňDoleva()`, nebo volání metody `posuňDoprava()` místo metody `posuňDoleva()`. Programátor poté obvykle zkontroluje všechny možné příčiny problému a chybu odhalí. Může se také stát, že ani jedna z možných příčin chybu nezpůsobuje – pak je skutečná příčina skrytá hlouběji v programu a ke slovu se dostanou pokročilé metody ladění, jako jsou například kontrolní výpisy nebo debugger.

2.1.3 JAK CHYBY OPRAVIT

Pokud programátor zjistí příčinu chyby, pak je obyčejně způsob opravy již zřejmý. Následuje poté pouze úprava kódu. Ovšem i během ní je nutné si dávat pozor, neboť úpravou taky můžete zavléci nebo způsobit jinou chybu.

Aby vaše oprava chyby byla co nejkvalitnější a nejúspěšnější, seznámím vás s několika zásadami při opravě chyb (podle knihy *Dokonalý kód* od Steve McConnella):

- **Nejprve problém pochopte, pak ho opravte:** Programu, chybě a její příčině musíte rozumět, abyste mohli něco modifikovat. Pokud vám není jasné, co chybu způsobuje nebo dokonce jak program pracuje, musíte nejprve zlepšit svoje pochopení programu. Pokud budete rozumět kontextu chyby (chápejte – programu), snadněji naleznete její příčinu a budete mít větší jistotu při opravování. Opravováním chyb tzv. „naslepo“ spíše více chyb přiděláte, než abyste nějakou opravili.
- **Opravujte vždy jen jednu chybu zároveň:** Často se stane, že v programu zjistíte více chyb najednou. Opravte vždy jen jednu chybu a neopravujte další, dokud ta předchozí není opravená. Tento postup nemusí být vždy možný. Pak byste měli opravovat vždy pouze co nejmenší možnou část programu.
- **Opravu otestujte:** Po opravě otestujte, jestli byla vaše oprava úspěšná. Vyhnete se tak tomu, že budete muset celý cyklus hledání příčiny a opravy absolvovat znova poté, co vás již jednou opravená chyba po delším čase znovu překvapí.
- **Opravte problém a ne příznaky:** Pokud zjistíte, že se něco nechová tak, jak má, opravte příčinu, ne příznak. Jednoduchý příklad: máte metodu, která zkon-

troluje, jestli je zadané číslo prvočíslo. Víte, že pro číslo 17 metoda vrací `false`, i když sedmnáctka je prvočíslo. Chybná oprava je přidat do metody řádek:

```
if(číslo == 17) return true;
```

Správné řešení je analyzovat kód výpočtu, proč pro 17 vrací `false`.

2.2 TYPICKÉ CHYBY

V následujících odstavcích vás seznámím s jednotlivými typy chyb a uvedu, jak tyto chyby vznikají, jak je odhalit a jak je opravit.

2.2.1 SYNTAKTICKÉ CHYBY

Zde vám popíšu několik chyb, které může překladač při zpracování kódu vypsat:

2.2.1.1 not a statement

Tato chyba se vyskytne, když překladač očekává nějaký příkaz, avšak text na daném řádku příkaz nepředstavuje. Nejčastěji tento typ chyby zapříčiněn překlepy. Tuto chybu způsobí například následující kód:

```
String jmeno;
jmeno == "Pepa";
```

Překladač očekává operátor přiřazení, ale vy jste omylem napsali rovnítka dvě, což je operátor porovnání. Chyba se také může vyskytnout, pokud máte dlouhý příkaz rozdělený na několik řádků a některý řádek jste omylem ukončili středníkem.

2.2.1.2 ; expected

Typická chyba, jíž se dopouštějí i ti nejzkušenější programátoři. Vzniká, když překladač detekuje příkaz, který není ukončen středníkem. Ale pozor, chyba může vzniknout i z jiných příčin, například pokud špatně uzavřete závorky. Další typickou příčinou je, když spojujete řetězec z více částí na více řádcích a zapomenete napsat mezi řádky znaménko plus:

```
String zprava = "Pro vaše jméno " + jméno + " nemáte vytvořen účet. "
    "Chcete ho vytvořit nyní?";
```

2.2.1.3 illegal start of expression

Chyba, která vzniká téměř výlučně díky tomu, že máte v kódu špatně zapsány závorky (chybějící nebo přebývající složené závorky). Nejčastěji se stává to, že zapomenete metodu uzavřít závorkou a překladač si poté myslí, že další metody jsou uvnitř první metody, což není přípustné:

```
public class Cisla
{
    public int get1(){
        return 1;

    public int get2(){
        return 2;
    }
}
```

(Zde není metoda `get1()` uzavřena.)

2.2.1.4 class, interface or enum expected

Chyba, která souvisí s chybou předchozí. Vzniká, pokud vám v kódu jedna složená závorka přebývá. Překladač si poté myslí, že tato přebytečná závorka uzavírá definici třídy a metody napsané po této závorce poté způsobí tuto chybu. Závorka způsobující tuto chybu je obyčejně umístěna bezprostředně nad řádkem, který chybu ohlásil.

2.2.1.5 reached end of file while parsing

Chyba, která také souvisí s chybou předchozí. Vzniká, pokud vám v kódu chybí jedna složená závorka na konci souboru, která uzavře definici třídy.

2.2.1.6 cannot find symbol

Chyba vzniká, když překladač narazí na nějaký symbol (vlastnost, proměnnou, metodu, konstruktor, třídu…), ale tento symbol nezná nebo není schopen určit jeho původ. Chyba vzniká, pokud používáte proměnou/vlastnost/metodu, kterou jste ale nedefinovali, nebo třídu, kterou jste zapomněli importovat pomocí deklarace `import`. Chyba také může vzniknout překlepem v názvu (nejčastěji záměnou velkých/malých písmen). Pro konstruktory může vzniknout také tak, že voláte konstruktor, který není definován (například s jinými parametry). Pro metody může chyba také vzniknout tím, že metoda, kterou používáte, není viditelná (je `protected` nebo `private`). Pro metody je chyba svázaná s další chybou:

2.2.1.7 [metoda(parametry)] in [třída] cannot be applied to (parametry)

Tato chyba vzniká, pokud se pokoušíte zavolat metodu s jinou sadou parametrů, než s jakou byla definována. Například když máte metodu:

```
public String udelejNeco(int i, String s){
```

tak všechna následující volání budou chybná:

`udelejNeco()` žádné parametry

`udelejNeco("bla")` chybí první parametr

`udelejNeco("bla", 3)` parametry v nesprávném pořadí

`udelejNeco("bla", 1, 2)` příliš mnoho parametrů.

Správné volání vypadá takto:

```
udelejNeco(3, "bla")
```

2.2.1.8 incompatible types

Tato chyba vznikne, pokud se pokoušíte přiřadit data určitého typu do proměnné, která má jiný, nekompatibilní typ (například se pokoušíte uložit řetězec do číselné proměnné). Chyba také může vzniknout překlepem v konstrukci `if`, když se použije pouze jedno rovnítko:

```
if(číslo = 3);
```

výraz „`číslo = 3`“ se vyhodnotí jako typ `int`, nicméně `if` vyžaduje `boolean` a vznikne tato chyba.

Pozor!, pokud je daný typ proměnné `boolean`, chyba není vyhozena – viz dále kapitola 2.2.3.4Přiřazení v podmínkách.

Chyba se také objeví, pokud se v metodě snažíte vrátit hodnotu jiného typu, než s jakým byla metoda deklarována (viz dále chyba missing return statement).

2.2.1.9 invalid method declaration

Tato chyba vzniká, pokud nedeklarujete hlavičku metody správě. Toto se nejčastěji stává tak, že deklarujete metodu, ale nedeklarujete její návratový typ. Pokud nechcete, aby metoda něco vracela, použijte návratový typ `void`:

```
public void nicNevracim(){
```

Chyba se také objeví, pokud uděláte překlep v názvu konstruktoru. Jelikož poté není název konstruktoru shodný s názvem třídy, překladač ho identifikuje jako normální metodu a vyžaduje u něho návratovou hodnotu. S návratovými typy metod se váží ještě další chyby:

2.2.1.10 cannot return a value from method whose result type is void

Tato chyba vzniká, pokud se pokoušíte vrátit hodnotu z metody, jejíž návratový typ je `void` (tedy metoda je deklarována, že nic nevrací):

```
public void getJednicka(){
    return 1;
}
```

Pokud chcete, aby metoda vracela nějakou hodnotu, změňte její deklaraci. Pokud nechcete, aby metoda něco vracela, zrušte příkaz `return` nebo z něj odstraňte hodnotu.

2.2.1.11 missing return statement

Tato je opakem předchozí chyby: deklarujete, že metoda vrací hodnotu, ale pak v jejím těle žádnou hodnotu nevrátíte:

```
public int getJednicka(){
    //žádný kód
}
```

Pamatujte, že metoda buď hodnotu vrací, nebo hodnotu nevrací, a pokud vrací, tak vždy jenom jednoho typu. Nemůžete mít proto metodu, která vrací hodnotu, pouze pokud jsou splněny určité podmínky nebo která může vrátit hodnoty více datových typů.

Pro metody s návratovou hodnotou je nutné, aby všechny možné způsoby zpracování v metodě na konci narazily na příkaz `return`. Např. toto je chybně:

```
public int getPočetVěcí(){
      if(početVěcí > 0){
            return početVěcí;
      }
}
```

Protože když bude počet věcí ≤ 0, metoda neví, co má vrátit (a je jedno, že víte, že během běhu nikdy počet věcí nebude ≤ 0, to překladač vědět nemůže).

Tato metoda je také chybná:

```
  public int getPočetVěcí(){
      if(početVěcí > 0){
            return početVěcí;
      }
      else{
            return "Nejsou zde žádné věci";
      }
  }
```

protože jednou vracíte `int` a podruhé `String`. Pokud chcete vypsat počet věcí textově, je lepší rozdělit metodu do dvou, jednu s návratovou hodnotou `int` a druhou s návratovou hodnotou `String`:

```
public int getPočetVěcí(){
      return početVěcí;
}
public String vypišPočetVěcí(){
      if(početVěcí == 0){
            return "Nejsou zde žádné věci.";
      }
      else{
            return "" + početVěcí;
      }
}
```

2.2.1.12 non-static [jméno] cannot be referenced from a static context

Tato chyba vzniká, když se ze statického kontextu (statická metoda nebo statický inicializační příkaz/blok) pokusíte získat atribut instance nebo se pokusíte zavolat metodu instance. Například:

```
class Kolo{
      private Color barva;

      public static Color getBarva(){
            return barva;
      }
      atd…
```

Zde je nutné si uvědomit rozdíl mezi instančním a statickým kontextem. Každá instance dané třídy má svůj instanční kontext, tj. ty věci, která má odlišná od jiných instancí. Např. v příkladu výše je atribut `barva` v instančním kontextu, tj. různá kola (různé instance třídy `Kolo`) mohou mít různou barvu. Statický kontext je ale naopak sdílený pro všechny instance dohromady. Kdyby byl atribut `barva` statický, pak by všechna kola měla stejnou barvu. Když poté ve statické metodě `getBarva()` vracíme instanční atribut `barva`, metoda neví, které instance to má barva být.

Je ovšem možné tuto instanci poskytnout a barvu vrátit z poskytnuté instance, např.:

```
public static Color getBarva(){
      return něco.getKolo().barva;
}
```

Zde už máme konkrétní instanci třídy `Kolo` (z objektu `něco`) a její barvu již můžeme vrátit. (Předpokladem ovšem je, že ve statickém kontextu třídy `Kolo` je nějaký objekt `něco` s metodou `getKolo()`, která vrátí instanci třídy `Kolo`.)

2.2.1.13 [jméno třídy] is not abstract and does not override abstract method in [třída]

Tato chyba vzniká, pokud vaše třída neimplementuje abstraktní metodu ve třídě, jejíž je potomkem nebo neimplementuje všechny metody v rozhraní, které implementuje.

Např. pokud programujete třídu `Kruh`, která dědí (rozšiřuje, `extends`) od abstraktní třídy `Tvar`. `Tvar` má deklarovánu abstraktní metodu `getObvod()` s návratovou hodnotou `int`. (Metoda slouží tomu, aby bylo možno pro všechny tvary získat jejich obvod, a je abstraktní, protože se pro různé tvary počítá obvod jinak.) Tím, že jste rozšířili třídu `Tvar`, zdědili jste od ní také povinnost implementovat metodu `getObvod()`. Tj. ve vaší třídě musí být deklarována a implementována metoda `getObvod()` s návratovým typem `int`.

Pro kruh by mohla její implementace vypadat takto:

```
return 2 * Math.PI * průměr;
```

Pro čtverec takto:

```
return 4 * strana;
```

Pro úsečku takto:

```
return délka;
```

Pro bod takto:

```
return 0;
```

Alternativně můžete svoji třídu deklarovat abstraktní. Povinnost implementovat metodu `getObvod()` se poté přenese na vaše potomky. V tomto případě by to dávalo smysl, pokud byste programovali např. třídu `EnUhelnik`, která by představovala jakýkoliv n-úhelník.

2.2.2 BĚHOVÉ CHYBY

Běhové chyby se projeví tak, že program během svého zpracování tzv. „vyhodí chybu“, tj. v určitém okamžiku se program dostane do nestandardního stavu a vygeneruje výjimku. Některé tyto výjimky jsou zachytitelné a zpracovatelné.

K výjimkám se zde nebudu obšírněji vyjadřovat, pouze vás odkážu na literaturu. Ve skriptech od manželů Pavlíčkových je celá kapitola 12 věnována výjimkám a je dokonce přístupná na java.vse.cz (http://java.vse.cz/pdf/skripta-vyjimky.pdf). Nabízí popis nejčastějších výjimek a jejich příčiny (některé výjimky jsou popsány detailněji v jiných kapitolách).

V knize „Myslíme objektově v jazyku Java“ od R. Pecinovského jsou výjimky popsány do nejmenších detailů, a to shodou okolností také v kapitole 12. Nabízí obšírnější a detailnější výklad toho, jak s výjimkami pracovat, ale k jednotlivým výjimkám nenabízí příliš mnoho popisu.

Na oficiálních stránkách Javy je k dispozici také tutoriál k výjimkám (na adrese http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html) a jednotlivé výjimky si lze vyhledat v dokumentaci (http://java.sun.com/javase/6/docs/api/java/lang/Exception.html).

2.2.3 LOGICKÉ CHYBY

Logické chyby jsou chyby, které způsobíte vy sami tím, že program naprogramujete sice bez syntaktických nebo běhových chyb, program se poté ale nechová tak, jak má (resp. tak, jak od něj očekáváte). Spousta těchto chyb je specifických a nelze na ně uplatnit jednu univerzální poučku.

2.2.3.1 Program nedělá to, co si myslíme

Poměrně velice častá chyba, kdy programátor napíše program jinak, než jak to zamýšlel. Například zapomene zavolat nějakou klíčovou metodu nebo přiřadí do proměnné špatnou hodnotu.

Zde je vždy důležité si uvědomit, co by danou chybu mohlo způsobit. Je nutné analyzovat symptomy problémů a podle nich si uvědomit, co se při zpracování programu udělá špatně. Např. pokud `Tvar` při metodě `začerni()` z plátna zmizí, chyba bude v metodě `začerni()` nebo v některé metodě, kterou metoda `začerni()` volá. Můžeme si být téměř zcela jistí, že chyba nebude v metodě `posuňDoprava()` ani ve třídě `Barva`. Nebo například když metody `začerni()` i `otoč90()` samostatně fungují, ale pokud se zavolá `otoč90()` po `začerni()`, tvar se přebarví zpět na svoji původní barvu, je poměrně jasné, že to musí být metoda `otoč90()`, která se zde chová nestandardně.

Ještě horší chyby jsou chyby v návrhu. Chyba v návrhu je taková, když zjistíte, že se chyba nedá opravit, aniž byste museli změnit vnitřní strukturu programu – tj. to, jak je program navržen. Velice jednoduchá chyba v návrhu by například byla, pokud byste chtěli do metody přidat metodu `barvaZpět()`, která by změnila barvu `Tvaru` zpět na předchozí, a během programování byste zjistili, že si historii barev `Tvar`u vůbec neukládáte. V tomto případě byste si museli do třídy `Tvar` přidat atribut `minuláBarva` a tento správně nastavovat v metodě `setBarva()`.

2.2.3.2 Nekonečný / nulový cyklus

Cykly jsou užitečné, pokud chcete, aby program vykonal něco vícekrát. Program sám nepozná, kolikrát se má něco vykonat, musíte mu to říct. A když mu to řeknete špatně, tak cyklus nikdy nezačne nebo naopak nikdy neskončí.

Nekonečný cyklus je takový, kde jeho podmínka ukončení bude vždy taková, aby cyklus nikdy neukončila. Například cyklus

```
for(int i = 0; i < 10; i++){
    //uděléj něco
    i = 0;
}
```

bude přičítat donekonečna, neboť před každou iterací (= průchodem cyklu), se řídicí proměnná `i` vždy vynuluje, tj. nikdy nebude větší nebo rovno 10.

Nekonečné cykly se využívají tam, kde ani za běhu není jasné, kolikrát se bude cyklus opakovat (např. při komunikaci po síti – zde není jasné, kolik zpráv ze sítě obdržíme, neboť to nezáleží na nás). Cykly se pak běžně ukončují příkazem `break`. Záměrně nekonečný cyklus se běžně zapisuje jako `for(;;)` nebo jako `while(true)`.

Nulový cyklus je takový cyklus, který neproběhne ani jednou. Např.:

```
for(int i = 0; i > 10; i++){
    System.out.println(i);
}
```

Takovýto cyklus měl vypsat číslo od 0 do 9, ale nevypíše nic. V podmínce je překlep – je použito opačné znamínko. Při spuštění cyklu se přiřadí `i=0` a přejde se na podmínku. Ta vyhodnotí, že `i` není větší než 10 a cyklus ukončí.

Další typickou chybou bývá použití zkráceného zápisu bez složených závorek:

```
for(int i = 0; i < 10; i++)
    něco();
```

Pokud omylem za cyklus `for` přidáte středník, cyklus proběhne, aniž by cokoliv zpracoval a příkaz `něco()` proběhne pouze jednou:

```
for(int i = 0; i < 10; i++);
    něco();
```

S tím souvisí i další úskalí zkráceného zápisu. Pokud totiž budete chtít do cyklu přidat další příkaz, nesmíte to napsat takto:

```
for(int i = 0; i < 10; i++)
    něco();
    něcoJiného();
```

protože pak se na příkaz `něcoJiného()` již cyklus nevztahuje. Cyklus je nutné zapsat standardním zápisem:

```
for(int i = 0; i < 10; i++){
    něco();
    něcoJiného();
}
```

Toto úskalí zkráceného zápisu se týká také podmínek `if` a `if-else`, cyklu `for-each` a cyklu `while`.

2.2.3.3 ConcurentModificationException

S touto chybou se také můžete občas setkat. Také souvisí s cykly, tentokrát s cyklem `for-each`. (Tj. s cyklem, který prochází `Iterable` objekt, nejčastěji nějakou kolekci - instanci rozhraní `Collection` – např. `for(String jmeno : jmena)`.)

Jde o to, že během procházení dané kolekce cyklem nesmíte tuto kolekci změnit. Pokud ji změníte, vyhodí se ConcurentModificationException. Např. nesmíte udělat tohle:

```
for(String jmeno : jmena){
    System.out.println(jmeno);
    jmena.remove(jmeno);
}
```

Oprava této chyby je jednoduchá, pokud je daná kolekce implementací rozhraní `List` – daný cyklus převeďte na klasický `for` cyklus:

```
for(int i = 0; i < jmena.size(); i++){
    System.out.println(jmena.get(i));
    jmena.remove(i);
}
```

Pro rozhraní `Set` není tato operace zcela jednoznačná, neboť neumožňuje získání elementů pomocí jejich indexů. Například byste mohli vypsání a vymazání rozdělit do dvou příkazů:

```
for(String jmeno : jmena){
    System.out.println(jmeno);
}
jmena.clear();
```

Nejkorektnějším způsobem je pak použití iterátoru:

```
Iterator<String> iterátor = jmena.iterator();
while(iterátor.hasNext()){
    System.out.println(iterátor.next());
    iterátor.remove();
}
```

Ne všechny implementace iterátoru ale příkaz `next()` podporují – některé při jejich použití vyhodí `UnsupportedOperationException`.

2.2.3.4 Přiřazení v podmínkách

Jedna z nejzákeřnějších chyb je použít pouze jedno = pro `boolean` proměnné v podmínkách, např.:

`if(jePlny = true)` nebo `if(jePlny = batoh.jePlny())`

Tato chyba vzniká, když omylem uvedete jedno rovnítko místo dvou rovnítek. Překladač v tomto případě nenahlásí žádnou chybu, ale program se začne chovat chybně (místo porovnání dojde k přiřazení hodnoty na pravé straně do proměnné na levé straně a tato hodnota se poté použije pro vyhodnocení příkazu `if`). Pokud nevíte, kde přesně máte hledat, je tato chyba velmi obtížně odhalitelná.

2.2.3.5 == a metoda equals(), HashSet

Tato konstrukce již byla několikrát omílána v učebnici a jistě i na přednášce. Je ale velice důležitá, tak si jí zde zopakujeme s tím, co její nedodržení způsobí za chybu.

Základní rozdíl mezi `==` a `equals()` je ten, že operátor `==` porovnává, zda jsou oba operandy shodné. Pro hodnotové typy (`int`, `boolean`, `double`, atd…) porovnává, jestli mají oba stejnou hodnotu. Pro odkazové typy (objekty) porovnává, zda oba odkazují na stejný objekt, tj. na stejnou jednu konkrétní instanci.

Např. si představte, že máte třídu `Kolo`, kde se kola neliší ničím jiným než svou barvou. Dále se koukněte na následující kód, který vytváří kola:

```
Kolo kolo1 = new Kolo("červená");
Kolo kolo2 = kolo1;
```

V tomto případě by `kolo1 == kolo2` vrátilo `true` – oba odkazy kolo1 i kolo2 ukazují na jeden a ten samý objekt. Pokud byste ale druhý řádek zaměnili za

```
Kolo kolo 2 = new Kolo("červená");
```

pak již příkaz `kolo1 == kolo2` vrátí `false`, neboť každá proměnná již ukazuje na jiný objekt, a nezáleží na tom, že obě kola jsou úplně stejná.

Metoda `equals()` v podstatě dělá to samé – v defaultní implementaci ve třídě `Object` porovnává instance. Rozdíl je v tom, že metoda `equals()` se dá přetížit – a implementovat zcela jiný systém pro porovnávání objektů. Např. třída `String` má tuto metodu přetíženou a porovnává obsah daného řetězce, ve třídě `Integer` se porovnávají uložená čísla, atd. Pro výše uvedenou třídu `Kolo` by se dala metoda `equals()` přetížit tak, aby porovnávala barvy kol.

Neříkám, že byste měli porovnávat vždy pomocí metody `equals()`, protože někdy se porovnávání instancí hodí. Měli byste si ale dávat pozor a rozlišovat mezi těmito dvěma možnostmi. Nejvíce chyb se dělá při porovnávání `String`ů. Lidé si myslí, že když napíší

```
jméno == "pepa"
```

tak porovnávají obsah řetězců. Proměnná `jméno` sice může obsahovat řetězec `"pepa"`, ale objekt to může být zcela jiný a operátor pak vrátí `false`.

Dalším kamenem úrazu je kolekce `HashSet` (ale problém se obecně týká všech implementací rozhraní `Set`), která nedovoluje vložení více shodných prvků, a tato shodnost je vyjádřena právě metodou `equals()`. Situace je sice o něco složitější, vás ale může zajímat jen to, že nesmíte udělat například tohle:

```
HashSet<String> jména = new HashSet<String>();
jména.add("Pepa");
jména.add("Jirka");
jména.add("Pepa");
```

V tomto případě se při druhém přiřazení hodnoty `"Pepa"` usoudí, že už jeden Pepa v kolekci je a do kolekce přidán nebude. Při výpisu kolekce:

```
for(String jméno : jména){
    System.out.println(jméno);
}
```

se tedy vypíše pouze jeden Pepa a jeden Jirka, a ne dva Pepové a jeden Jirka, jak by se mohlo na první pohled zdát.

Pozn.: pro řetězce může někdy operátor == fungovat. Je to zapříčiněno optimalizátorem, který totožné řetězce zapsané jako literály (tj. natvrdo v kódu) nevytváří vícekrát, ale místo toho vždy odkáže na jeden daný řetězec. Obecně by ale bylo chybou na tuto funkci v kódu spoléhat.

2.2.3.6 NullPointerException

Tuto výjimku vám program vyhodí, když chcete udělat nějakou operaci s odkazovou proměnnou, která však neukazuje na žádný objekt (tj. ukazuje na `null`). Nejčastěji se tato chyba stává, když si ve třídě deklarujete nějaký atribut, který ale následně zapomenete inicializovat (tj. přiřadit mu hodnotu). Další častou příčinou je, že používáte nějakou metodu, která může vrátit `null`, ale s tímto už dále v kódu nepočítáte:

```
/* může vrátit null pokud není nikdo přihlášen */
String jmeno = getPrihlasenyUzivatel();
/* může vyhodit NullPointerException, pokud není nikdo přihlášen */
jmeno.toLowerCase();
```

Problémem této chyby je, že pokud se vyskytne na nějakém řádku, kde se děje několik věcí, nelze jednoznačně určit, která proměnná zrovna byla `null`:

```
uzivatele.add(getPrihlasenyUzivatel().getJmeno().toLowerCase());
```

Pokud se na tomto řádku vyskytne `NullPointerException`, může to být proměnná `uzivatele`, výsledek metody `getPrihlasenyUzivatel()` nebo výsledek metody `getJmeno()`, co mohlo výjimku způsobit. Chybová hláška vám ale neřekne, která to byla, na to musíte přijít sami.

Jedním z postupů, jak tohoto dosáhnout, je rozdělit si složitý příkaz na jednotlivé elementární příkazy:

```
Uzivatel prihlasenyUzivatel = getPrihlasenyUzivatel();
String jmeno = prihlasenyUzivatel.getJmeno();
String jmenoMalymiPismeny = jmeno.toLowerCase();
uzivatele.add(jmenoMalymiPismeny);
```

Podle toho, na kterém řádku pak dojde k `NullPointerException` pak poznáte, která proměnná byla `null`. Pokud dojde na řádku 2, pak metoda `getPrihlasenyUzivatel()` vrátila `null`. Pokud na řádku 3, pak metoda `getJmeno()` vrátila `null`. Pokud na řádku 4, pak proměnná `uzivatele` má hodnotu `null`.

2.2.3.7 Uživatelský vstup

Tato chyba je spíše již teoretická pro vstupní kurzy, protože se s ní nesetkáte přímo, ale je dobré, abyste věděli, že existuje a že představuje nejenom chybu, ale v některých případech také bezpečnostní riziko.

Zde jde o to, že uživatelé interagují s programem pomocí ovládacích prvků (vstupní pole, vstupní formuláře, databáze, atd.) Tento vstup ale nemusí být vždy korektní. Např. když programujete textovou adventuru, tak uživatel jako příkaz může napsat nějaký nesmyslný řetězec, popřípadě příkazu může zadat nesmyslné parametry. Pokud byste tento vstup nevalidovali, tak byste zjistili, že se pokoušíte vykonat příkaz, který není definovaný, nebo obsloužit parametr, který nedává smysl.

Proto existuje uživatelská validace. Než program začne pracovat s uživatelským vstupem, měl by si zkontrolovat, že tento vstup dává smysl (např. že příkaz existuje). Při validaci pak záleží na vás, jak uživatelovi řeknete, že zadal neplatné příkazy. Běžnou praxí

pro programy s grafickým rozhraním je zobrazit dialogové okno, pro textové rozhraní vypsat chybovou hlášku přímo pod příkaz.

Chyba dostává ještě jiný rozměr v případě, že uživatelský vstup používáte přímo v příkazech pro databázi. Pokud tento vstup nevalidujete, vystavujete se možnosti útoku pomocí SQL injection (více na http://cs.wikipedia.org/wiki/SQL_injection).

2.2.3.8 Použití překrytelné metody v konstruktoru

Jedna z nejzákeřnějších chyb, které můžete ve svých programech udělat. Překrytelná metoda je taková metoda, kterou mohou potomci zdědit a překrýt vlastní verzí metody. Jsou to všechny metody, které nejsou `private` a zároveň nejsou `final`. Ačkoliv použití takovéto metody v konstruktoru není chybou samo o sobě, představuje rizikové místo, které může vyvolat chybu až dlouho poté, co byla třída naprogramována (zpravidla v tu nejméně vhodnou dobu). Jde o to, že když použijete v konstruktoru překrytelnou metodu, nikdy nevíte, kdo vaší třídu zdědí a co budou s tímto potomkem dělat, jaké metody překryjí a co v těchto metodách budou dělat.

Zde je příklad: Máte třídu `Ctverec`, který se po vytvoření ihned namaluje na plátno (čísel si zatím nevšímejte):

```
class Ctverec
{
	int strana;
	Platno platno;

	Ctverec(int strana, Platno platno){
		2) this.strana = strana;
		3) this.platno = platno;
		4) namaluj();
	}
	void namaluj(){
		platno.namalujCtverec(strana);
	}
}
```

a pak máte třídu barevný čtverec, který k tomuto základnímu čtverci přidává barvu:

```
class BarevnyCtverec extends Ctverec
{
	Barva barva;

	BarevnyCtverec(Platno platno, int strana, Barva barva){
		1) super(strana, platno);
		6) this.barva = barva;
	}
	void namaluj(){
		5) platno.namalujCtverec(strana, barva);
	}
}
```

Zde je si důležité uvědomit, jak jdou postupně příkazy za sebou. Já jsem příkazy očísloval, jak jdou za sebou při zavolání konstruktoru `BarevnyCtverec(Platno platno, int strana, Barva barva)`.

Zde si všimněte, že došlo k chybě tím, že k malování (příkaz 5) dojde ještě před tím, než se přiřadí barva (příkaz 6), tj. čtverec se namaluje bez barvy. (Což např. může vyvolat `NullPointerException` v metodě `namalujCtverec()`, protože se v parametru `barva` předá hodnota `null`.)

2.2.3.9 IndexOutOfBoundsException

Tato chyba vzniká, pokud chcete přistupovat k prvku pole nebo řetězce na indexu, který neexistuje. Toto je téměř vždy zapříčiněno tím, že si neuvědomíte, že první prvek má index 0 a poslední prvek má index délka-1. Příklad pole:

Prvek	"Jana"	"Lída"	"Adéla"	"Klára"	"Kristýna"
Index	0	1	2	3	4

Ačkoliv dané pole má 5 prvků (tj. `length` = 5), prvek na pozici 5 neexistuje, poslední prvek má index 4. Pokud chcete přistupovat k poslednímu prvku pole bez ohledu na délku, musíte napsat `length-1`. S tím také souvisí cyklus pro procházení polí:

```
for(int i = 0; i <= pole.length; i++)
```

je špatně, protože pole projde od 0 do `length`.

Správně:

```
for(int i = 0; i < pole.length; i++)
for(int i = 0; i <= pole.length - 1; i++)
```

Chyba také vzniká, pokud používáte metodu `substring()` na řetězci. Např. pokud chcete získat celý `String` kromě prvního písmena, musíte napsat `string.substring(1, string.length-1)`.

2.3 JAK LADIT (DEBUGGING)

Chyba se vždy projeví tím, že se program chová jinak, než zamýšlíte. Jak jsem již uvedl, měli byste začít vyšetřovat příčinu chyby. Pokud na příčinu nepřijdete (nebo jste líní analyzovat hromadu kódu příkaz po příkazu), musíte začít program ladit (debuggovat).

Dva nejběžnější ladící postupy jsou logování a programový debugging. Zatímco logování je jednodušší forma a používá se pro odhalení jednoduchých chyb, programový debugging je mocný a sofistikovaný nástroj, se kterým lze odhalit téměř všechny chyby (ale na malé chyby se nehodí – je to jako vzít příslovečný kanón na mouchu).

Ladění má za cíl poskytnout programátorovi stav zpracování programu v určitých daných časových okamžicích. Programátor tento skutečný stav musí porovnat se stavem chtěným, a pokud se tyto dva stavy liší, měl by program opravit, aby se tyto stavy již nelišily. Znovu zde ale zdůrazňuji, že svému programu musíte opravdu rozumět. Pokud nevíte, jak se má program v určitém okamžiku chovat, nevíte ani, jestli je daný stav správný nebo ne.

2.3.1 RUČNÍ PROCHÁZENÍ KÓDU

Tato technika se dá použít pouze pro ty nejjednodušší chyby. Nejčastěji se používá, pokud znáte příčinu chyby již od začátku, pouze musíte v kódu nalézt dané inkriminované místo, které chybu vyvolává.

Pro složitější ladění je tato technika nevhodná. Zaprvé si nemůžete zapamatovat všechen relevantní kód, který s chybou může souviset. A za druhé je všeobecně známo, že člověk své chyby nevidí – je příliš ovlivněn svou představou *co by mělo být* a špatně od tohoto odlišuje *co skutečně je*.

2.3.2 LOGOVÁNÍ

Logování je jednoduchá ladící procedura spočívající v tom, že do kódu programu přidáte příkazy, které vypisují (nejčastěji na standardní výstup) stav programu. Tento výpis dokonce Java sama generuje v případě, že se v programu vyskytne neošetřená výjimka – místo toho, aby program spadl bez jakékoliv informace, napíše vám, co se stalo za chybu a kde.

Pokud ve svém programu narazíte na chybu, u které si nejste zcela jistí, co ji způsobuje, měli byste chybu zkusit odhalit logováním. Nejzákladnějším logovacím nástrojem jsou systémové hlášky `System.out.pritnln()`. Ty vám umožňují na standardní výstup zapsat jakoukoliv informaci a tyto zápisy dokonce po ukončení programu nezmizí (tj. pomocí nich můžete upravovat kód).

Je samozřejmě poněkud hloupé uvádět logovací hlášky za každý příkaz – zaprvé 99% příkazů nebude mít s chybou nic společného, a za druhé by těchto hlášek bylo takové množství, že byste se v nich stejně nevyznali. Dejte jednu, dvě či tři tyto hlášky na místa, která by mohla mít s chybou něco společného. Např. pokud chcete nakreslit na plátno `Tvar`, ale ten se při spuštění programu nenakreslí, umístěte do metody `nakresli()` třídy `Tvar` jednoduchou hlášku `System.out.println("Teď se kreslím")`. Po spuštění programu uvidíte, zda se tato hláška objeví (a problém bude ve třídě `Plátno`, které daný `Tvar` odmítlo nakreslit), nebo se neobjeví (a vy budete vědět, že je problém ve třídě `Tvar`, jehož metoda `nakresli()` není nikdy zavolána).

Pokud těchto hlášek dáváte více, měli byste vědět, v jakém pořadí budete očekávat jejich výpis. Pokud např. kreslíte tvary, ale jeden tvar bude ve výpisu chybět, měli byste začít zkoumat, proč se tento tvar nenakreslil, ale ostatní ano. Pokud chcete zkoumat zpracování cyklu, a v těle tohoto cyklu je více příkazů a více vypisujících hlášek, měli byste si na začátek cyklu umístit jednoduchou hlášku

```
System.out.println("-------------");
```

abyste pak ve výpisu jasně poznali, kdy začala nová iterace cyklu.

Do hlášek můžete samozřejmě dynamicky přidávat hodnoty proměnných. Např. místo jednoduchého `System.out.println("Teď se kreslím")`, můžete vypsat něco více smysluplného, jako např.:

```
System.out.println("Kreslím obdélník na souřadnicích [" + x + ", " + y +
"] a velikosti " + šírka + " na " + výška + " s barvou " + barva);
```

Výpis takové hlášky pak může vypadat např.: „Kreslím obdélník na souřadnicích [25, 90] a velikosti 80 na 120 s barvou červená“, což vám o zpracování kreslení obdélníku poví o mnoho více, než jednoduché „Teď se kreslím“.

2.3.3 STACK TRACE

Jak jsem již uvedl výše, pokud dojde v programu k neošetřené výjimce, program automaticky vypíše chybu na výstup. S touto chybou se vypíše i tzv. stack trace, tj. popis toho, jak se program k chybě dostal. Informace z tohoto stack trace často bývají dostačující k určení příčiny chyby. Výpis chyby se stack tracem může vypadat např. takto:

```
Exception in thread "main" java.lang.IllegalArgumentException: Takto
pojmenovanou barvu neznam.
    at tvary.Barva.getBarva(Barva.java:93)
    at tvary.Obdelnik.setBarva(Obdelnik.java:238)
    at tvary.Main.main(Main.java:13)
```

Takový výpis vám říká, že v příkazu `getBarva()` ve třídě `Barva` na řádku 93 došlo k chybě `IllegalArgumentException`. Do této metody se program dostal z příkazu na řádku 238 v metodě `setBarva()` ve třídě `Obdelnik`. A do této metody se program dostal z metody `main()` ve třídě `Main` na řádku 13.

Stack tracy bývají někdy desítky příkazů dlouhé, ale obvykle stačí zanalyzovat první dva nebo tři příkazy, abyste příčinu chyby objevili. Jakmile začnete programovat složitější programy, mohou se vám objevit výjimky, kde bezprostřední příčina není na prvním příkazu. Pak je nutné projet výpis chyby tak daleko, až narazíte na příkaz, který pochází z vašeho kódu. Tam je potřeba začít hledat.

2.3.4 VLASTNÍ LOGOVÁNÍ

Někdy vám hlášky `Sysytem.out.println()` nebo standardní chybový výstup nemusí stačit. Především někomu může vadit, že pro každou hlášku musí napsat `System.out.println()`, což je poměrně dlouhý příkaz. Hlavní nevýhodou ale je, že si nemůžete vybrat, které hlášky zobrazíte (prostě se zobrazí všechny) a kam tyto hlášky zobrazíte (všechny půjdou na standardní výstup). Pokud programujete nějaký větší projekt a všechny hlášky v programu necháte, za chvíli se vám jich bude generovat tolik, že se v nich nevyznáte. Nebo pokud chcete výpis hlášek poslat kamarádovi, nemusí se vám vždy hodit to kopírovat ze standardního výstupu (např. ICQ má omezenou délku zprávy), lepší by bylo, kdyby se hlášky zapsaly do souboru, který pak kamarádovi pošlete.

Proto vám doporučuji vytvořit jednoduchou třídu `Log`:

```
public class Log
{
	public static void log(String log){
		System.out.println(log);
	}
}
```

Tato třída má několik výhod. Zaprvé, je jednodušší a kratší psát `Log.log()` než `System.out.println()`. Dále kdykoliv budete chtít např. začít logovat do souboru, stačí si tuto třídu upravit, aby zprávy posílala také do souboru. Když budete chtít logovat jenom některé zprávy, stačí si do metody `log()` přidat parametr, který určuje skupinu zprávy. Ve třídě `Log` pak nastavíte, která skupina zpráv se má vypisovat. Více skupin má také tu výhodu, že pokud pracujete v týmu, můžete si přiřadit skupiny a pak se vám budou vypisovat jenom vaše zprávy a ne zprávy ostatních.

V tomto případě jsou možnosti téměř neomezené – se třídou `Log` si můžete dělat, co se vám zlíbí a co se vám zrovna hodí. Např. při odevzdání programu již nebudete chtít tyto zprávy vypisovat. Není nic jednoduššího, než je v kódu zakázat:

```
public class Log
{
	private static final boolean logujeme = false;

	public static void log(String log){
		if(logujeme == true){
			System.out.println(log);
		}
	}
}
```

Takový kód už program ani nezpomalí – již v době překladu je jasné, že metoda `log()` nikdy nic dělat nebude, tak optimalizátor její volání ze zpracování vynechá. Pokud byste jen přeci potřebovali opět hlášky zapnout, stačí změnit atribut `logujeme` na `true`.

Nevýhodou této třídy je, že ji musíte pokaždé importovat (což `System.out.println()` nevyžaduje)

Pokud nejste ochotní nebo neumíte naprogramovat vlastní logovací třídu, můžete použít již hotové logovací frameworky. Ačkoliv logovací frameworky se v konceptu neliší od vlastní logovací třídy, jsou již dotažené do konce profesionálními programátory a cokoli od nich potřebujete, je pravděpodobně v nich již naprogramováno. Existují dva nejpoužívanější: zabudované Java Logging API a Apache Log4J.

2.3.5 ASSERT

Příkaz `assert` byl do Javy přidán s verzí 1.4. Je to přímo konstrukce v jazyce Java, která vám umožní porovnávat danou hodnotu s `true` nebo `false`. Příklad:

```
assert uživatel.getJmeno().equals("Pepa");
```

Pokud je příkaz za slovem `assert` vyhodnocen jako `true`, program pokračuje dál. Pokud je vyhodnocen jako `false`, program vygeneruje výjimku `AssertionError`, která, pokud není zachycena, zastaví chod programu.

Pokud na vhodná místa programu dáte takovéto aserce, pak je odhalení nové chyby potencionálně mnohem jednodušší. I když si chybu zanesete již do otestovaného kódu, nebo by se chyba projevila pouze nepřímo, aserce vás v tomto případě ihned upozorní, že něco není v pořádku a kde to není v pořádku. Ušetří vám tak mnoho času testováním.

Spíše než ladící nástroj se aserce používají jako nástroj pro kontrolu kvality. Kdykoliv program spustíte, mohou aserce pracovat jako pojistka proti nebezpečným činnostem, které by např. mohly poškodit uživatelská data. Defaultně je zpracování asercí v Javě vypnuto.

2.3.6 LADĚNÍ S POMOCÍ DEBUGGERU

Mnoho integrovaných vývojových prostředí poskytuje vývojářům debugger. Debugger je vestavěná utilita, která umožňuje programátorům zkoumat svůj program a řídit jeho zpracování v reálném čase za běhu.

I BlueJ obsahuje tento nástroj, i když primitivnější než jaký mají profesionální vývojová prostředí. Jeho použití je popsáno v povinné literatuře a i na Internetu (v BlueJ tutoriálu na straně 27 – viz. http://www.bluej.org/tutorial/tutorial-201.pdf). Jeho praktické použití najdete demonstrované na některých videích o kousek dole.

2.4 ODHALENÍ NEJČASTĚJŠÍCH CHYB

Zde najdete několik videí,[2] která vám přiblíží postup ladění na několika nejčastějších chybách. Projekt je ke stažení zde.

2.4.1 PROGRAM NEDĚLÁ, CO SI MYSLÍME

Uvažujme, že chceme přidat do projektu nový tvar s názvem `Kříž`, který by nakreslil na plátno kříž. Třída si vytvoří dva stejné obdélníky a bude je malovat kolmo na sebe.

Scénář:

1. Uživatel vytvoří novou třídu `Kriz` a implementuje metody a konstruktory.
   V bezparametrickém konstruktoru bude uvedena červená barva. Nicméně, při vytváření obdélníků zapomene uvést jejich barvu (zavolá konstruktor bez barvy), a nakreslí se tedy s defaultní barvou ze třídy `Obdélník`, což je červená.
2. Uživatel zkusí program spustit, a ten bude fungovat.
3. Uživatel změní v bezparametrickém konstruktoru barvu na modrou. Program spustí, ale nebude fungovat (kříž bude stále červený).

[2] Pro textovou verzi této práce jsou videa uvedena ve formě jejich krátkého scénáře. Samotná videa jsou přílohou elektronické verze.

4. Uživatel bude debuggovat třídu `Kříž`. Nastaví zarážku na bezparametrický konstruktor. Program spustí a prochází jednotlivé příkazy.
5. V metodě `nakresli()`, kde se vytváří obdélníky, zjistí, že se obdélníky vytváří s červenou barvou.
6. Program ukončí a začne hledat příčinu. Zjistí, že se obdélníkům vůbec informace o barvě nepředává, protože se volá špatný konstruktor.
7. Chybu opraví a program spustí. Program funguje.

2.4.2 OPERÁTOR PŘIŘAZENÍ V PODMÍNCE

Chceme přidat do třídy `Kříž` atribut, který bude specifikovat, zdali se má kříž nakreslit normálně nebo šikmo (tj. jako + nebo jako ×). Metodu `nakresli()` musíme podle toho pozměnit.

Scénář:

1. Uživatel doplní do třídy `Kříž` atribut `šikmý`, který bude určovat, jak se má daný kříž nakreslit. Atribut se správně inicializuje podle parametru v konstruktoru.
2. Upraví metodu `nakresli()` tak, aby kreslila šikmé kříže šikmě. Při psaní podmínky ale udělá překlep, místo aby porovnával hodnotu atributu `šikmý` s hodnotou `true`, tuto hodnotu do atributu přiřadí.
3. Uživatel zkusí vytvořit šikmý kříž. Program funguje
4. Uživatel zkusí vytvořit normální (nešikmý) kříž. Vytvoří se však znovu šikmý kříž.
5. Uživatel bude debuggovat metodu `nakresli()` ve třídě `Kříž`. Zkusí znovu vytvořit normální kříž. Program se zastaví v metodě `nakresli()`. Je vidět, že atribut `šikmý` je `false`, tj. že se inicializoval správně.
6. Nechá metodu projít až k podmínce, kde zjistí, že se hodnota atributu `šikmý` najednou změnila. Při bližším pohledu uživatel zjistí, že v podmínce je pouze jedno rovnítko.
7. Chybu opraví přidáním druhého znaménka rovná se. Zkusí znovu vytvořit rovný kříž a program nyní funguje správně.

2.4.3 NULOVÝ CYKLUS

Uživatel si chce vytvořit novou třídu – hada. Had bude reprezentován novou třídou `Had`. Had bude tvořen po sobě jdoucími kruhy. Počet kruhů uživatel specifikuje v konstruktoru.

Scénář:

1. Uživatel vytvoří novou třídu.
2. Uživatel naprogramuje třídu, ale v metodě `nakresli()` otočí v cyklu `for` znaménko porovnání.

3. Uživatel třídu vyzkouší, ale žádný had se nenakreslí. Uživatel rovněž zkusí zavolat metodu `nakresli()` ručně, ale ani to hada nenakreslí.
4. Uživatel si dá zarážku na metodu `nakresli()`, poté začne krokovat.
5. Během krokování zjistí, že cyklus neprošel ani jednou. Uživatel při bližším pohledu odhalí, že otočil znaménko.
6. Chybu opraví. Třídu vyzkouší a ta nyní funguje.

2.4.4 ZÁMĚNA == A EQUALS()

Uživatel vytvoří třídu `Kreslič`. `Kreslič` bude knihovní třída, která ze zadaného souboru načte názvy tvarů k namalování a tyto tvary poté namaluje.

Scénář:

1. Uživatel udělá novou třídu – kresliče. Tento kreslič bude podle souboru kreslit tvary. V souboru bude uveden na každém řádku uveden název tvaru.
    a. Pokud kreslič narazí na řádek, kde není uveden podporovaný název tvaru, vypíše, že zadaný tvar neumí nakreslit.
2. Uživatel vytvoří soubor se třemi tvary (které kreslič umí nakreslit) a spustí program. Program vyparsuje soubor, ale žádné tvary nenakreslí. Ve výpisu se objeví, že kreslič dané tvary neumí nakreslit.
3. Uživatel otevře třídu kresliče a zanalyzuje příkazy porovnání. Zjistí, že se názvy porovnávají příkazem `==` a ne metodou `equals()` a že takováto podmínka nemůže být nikdy splněna.
4. Uživatel zamění operátor `==` za metodu `equals()` a poté zkusí tvary znovu vykreslit. Třída již nyní funguje.

2.4.5 METODA EQUALS() A TŘÍDA HASHSET

Uživatel změní implementaci třídy kresliče tak, aby si tvary ukládala do `HashSetu` a vykreslila je až poté, co jsou všechny načtené.

Scénář:

1. Uživatel změní implementaci třídy kresliče. Nyní si ukládá názvy tvarů do `HashSetu`.
2. Uživatel zkusí spustit kresliče na souboru, který má tři obdélníky. Nakreslí se pouze jeden obdélník.
3. Uživatel si třídu `Kreslič` doplní o ladící hlášky `System.out.println()`. Po průběhu programu hlášky zanalyzuje a zjistí, že se tvary špatně ukládají do kolekce.
4. Uživatel změní implementaci kolekce na takovou, která již na metody `equals()` nebere ohled – např. `ArrayList`.
5. Uživatel zkusí třídu spustit znovu – nyní třída funguje.

2.4.6 PŘEKRYTELNÁ METODA V KONSTRUKTORU

Uživatel vytvoří novou třídu `SpojitýHad` (rozšíření třídy `Had`), který bude reprezentovat hada, jehož tělo je spojeno obdélníkem (takže bude vypadat spojitě). Třída bude obsahovat extra parametr spojitost, který bude udávat, jak má být tento obdélník široký (a tedy jak má být had spojitý).

Scénář:

1. Uživatel vytvoří třídu.
2. Uživatel naprogramuje třídu. Do konstruktoru nedá metodu `nakresli()` – protože ta už je v konstruktoru třídy `Had`.
3. Pokusí se třídu spustit s nenulovým parametrem spojitosti. Had se stejně vytvoří nespojitý.
4. Uživatel bude krokovat. Umístí zarážku do konstruktoru. Během chodu uživatel zjistí, že metoda `nakresli()` se volá ještě před inicializací proměnné spojitost.
5. Uživatel zruší volání metody `nakresli()` ve třídě `Had` a přesune ho do třídy `SpojitýHad`.
6. Nyní zkusí spojitého hada vytvořit. Třída funguje.

2.4.7 PŘEKRYTELNÁ METODA V KONSTRUKTORU – VYHOZENÍ VÝJIMKY

Uživatel si bude chtít vytvořit novou třídu – dvojbarevného hada. Dvojbarevný had se bude kreslit jako normální had, ale s dvěma barvami (barvy se budou střídat mezi jednotlivými koly hada).

Scénář:

1. Uživatel vytvoří třídu `DvojbarevnýHad`.
2. Uživatel naprogramuje třídu. Do třídy přidá atribut `barva2`. Do konstruktoru nedá metodu `nakresli()` – protože ta už je v konstruktoru hada.
3. Uživatel spustí třídu. Program spadne na `NullPointerException` v metodě `nakresli()`. Uživatel zadá zarážku do metody `nakresli()`.
4. Uživatel krokuje. Během zpracování programu zjistí, že je parametr pro druhou barvu neinicializovaný.
5. Zjistí, že metoda `nakresli()` se volá před inicializací proměnné `barva2`, protože překryl metodu, která se používá v konstruktoru třídy `Had`.
6. Uživatel zruší volání metody `nakresli()` ve třídě `Had` a přesune ji do třídy `DvojbarevnýHad`.
7. Nyní zkusí dvojbarevného hada vytvořit. Třída funguje.

3 ZÁVĚR

3.1 ZHODNOCENÍ

Cíle práce tak, jak byly v úvodu stanoveny, se podařilo beze zbytku naplnit. Posluchačům úvodních kurzů programování se dostává do ruky příručka ladění, která alespoň v nejzásadnějších bodech osvětluje problematiku chyb a ladění. Jestli se podaří naplnit i přínos práce, to bohužel ukáže až čas.

Práce má samozřejmě mnohé rezervy. Například se téměř vůbec nevěnuje problematice testování, což je činnost, která s laděním velice úzce souvisí. Teoretické zázemí ladění by mohlo být také lépe pokryto, ale tento nedostatek je vyvážen velikým množstvím praktických příkladů a doporučených postupů. Nedílnou součástí úspěšného vývoje softwaru je také správná metodika vývoje a zejména postupy, které vytváření chyb v programu co nejvíce zamezují. I tento aspekt vývoje není v této práci popsán.

3.2 VYSVĚTLIVKY

Pojem	Zkratka	Vysvětlení[3]
Application Programming Interface	API	Souhrn metod, které operační systém, program, knihovna či třída nabízí programátorům k použití.
BlueJ		IDE pro Javu speciálně navržené pro výuku Javy.
Grafické uživatelské rozhraní	GUI	Jedna z možností, jak může uživatel interagovat s programem – pomocí grafického rozhraní zobrazeného na monitoru počítače.
Integrated Development Environment	IDE	Integrované vývojové prostředí. Software pro vytváření jiného software.
Java Development Kit	JDK	Klientský program pro programátory v Javě. Obsahuje v sobě JRE plus vývojové nástroje a pomocné programy pro vývoj softwaru.
Java Runtime Environment	JRE	Klientský program, který je nutno instalovat, aby mohly programy napsané v Javě běžet na počítači. Obsahuje JVM, standardní knihovnu a další programy.
Java Virtual Machine	JVM	Platforma, na které jsou postaveny všechny programy napsané v Javě.
Objektově orientované programování	OOP	Metoda vývoje softwaru, která program reprezentuje jako soubor objektů, které si mezi sebou posílají zprávy
Unified Modelling Language	UML	Formální definice jazyka používaného k modelování architektury softwaru.

[3] Zdroj: vše autor

3.3 SEZNAM ZMĚN V BLUEJ

3.3.1 JAVAC.HELP

Chyba	Původní text	Nový text
not a statement (To není příkaz)	Napsali jste kód, který není příkazem. Zkontrolujte znovu tento řádek.	Kód na tomto řádku není příkazem. Pravděpodobně jste udělali překlep v příkazu.
illegal start of expression (Zakázaný začátek výrazu)	Na této pozici, je očekáván počátek výrazu, ale je zde něco jiného. Pravděpodobně vám v předchozím kódu něco přebývá nebo chybí. Zkontrolujte správnost definice typu.	(Špatný začátek výrazu) Na tomto řádku je neočekávaný výraz. Zkontrolujte si, zda máte všechny složené závorky patřičně uzavřeny na správných pozicích.
class, interface, or enum expected	(chybí – nebyl uveden)	(Očekávaná deklarace třídy, rozhraní nebo výčtového typu) Na označené pozici došlo k uzavření třídy a další kód je špatně rozpoznán. Pravděpodobně vám přebývá jedna složená závorka.
reached end of file while parsing	(chybí – nebyl uveden)	(Při parsování narazil překladač na konec souboru) Překladač neočekávaně narazil na konec souboru. Pravděpodobně vám chybí jedna složená závorka na konci souboru, která uzavře definici třídy.
cannot find symbol * (Nelze najít symbol…)	Na označeném místě používáte symbol (jméno proměné, třídy nebo metody), která není deklarováná v žádné viditelné oblasti. Zkontrolujte, zda jste neudělali překlep nebo zda jste danou proměnnou nezapomněli deklarovat. Objevíte-li chybu, napravte ji.	Na označeném místě používáte symbol (jméno proměnné, třídy nebo metody), která není deklarovaná v žádné viditelné oblasti. Zkontrolujte, zda jste neudělali překlep nebo zda jste danou proměnnou nebo metodu nezapomněli deklarovat.
* cannot be applied to * (… operátor či metoda nemůže být použit (a) pro…)	Operátor, resp. metoda, kterou se zde pokoušíte použít, nemůže být pro tento datový typ použit (a). Používáte buďto špatný typ dat, nebo nevhodný operátor. Chyba je často vyvolána tím, že použijete takovou sadu parametrů, pro kterou není definována potřebná přetížená verze dané metody.	Tato chyba byla rozdělena na dvě: - operator * cannot be applied to * (… operátor nemůže být použit pro…) Operátor, kterého se zde pokoušíte použít, nemůže být pro tento datový typ použit. Používáte buďto špatný typ dat, nebo nevhodný operátor. (Např. se pokoušíte násobit Stringy.)

Chyba	Původní text	Nový text
		- * cannot be applied to * (... metoda nemůže být použita pro...) Metoda, kterou se zde pokoušíte použít, je deklarována s jinou sadou parametrů. Metodu se snažíte volat buď s příliš málo parametry, s příliš mnoho parametry, s parametry jiných typů nebo se správnými parametry ve špatném pořadí.
incompatible types * (Nekompatibilní typy...)	Na tomto místě je očekáván jiný datový typ. Přiklad: uvedli jste typ "String" na místě, kde je očekáván typ "int".	Na tomto místě je očekáván jiný datový typ. Například jste uvedli typ "String" na místě, kde je očekáván typ "int". Je také možné, že v metodě vracíte hodnotu jiného typu, než jaký je deklarován v hlavičce metody.

Dále byly opraveny některé překlepy.

3.3.2 EXCEPTION.HELP

Vyjímka	Původní text	Nový text
ClassCastException	Chyba při přetypování. Příklad: Object nejakyOdkaz; ..... (MojeTrida)nejakOdkaz Odkaz v proměnné "nejakyOdkaz" je přetypován na typ MojeTrida. To je však dovoleno pouze tehdy, odkazuje-li na instanci třídy MojeTride nebo některého z jejích potomků.	Chyba při přetypování. Příklad: Object nejakyOdkaz; ..... MojeTrida mojeTrida = (MojeTrida)nejakyOdkaz Odkaz v proměnné "nejakyOdkaz" je přetypován na typ MojeTrida. To je však dovoleno pouze tehdy, odkazuje-li na instanci třídy MojeTrida nebo některého z jejích potomků.
IllegalArgumentException	Nápověda není k dispozici.	Tato výjimka je vždy vyhazována manuálně některou třídou, kterou v projektu používáte. Označuje, pokud jste nějaké metodě poskytly parametry, se kterými nepočítala. Chybové hlášení u této chyby by vám mělo napovědět, co je špatně.

Vyjímka	Původní text	Nový text
IOException	(výjimka nebyla uvedena)	Chyba, která indikuje, že se něco nepovedlo se vstupem nebo výstupem. Například se pokoušíte číst neexistující soubor nebo se pokoušíte zapisovat do souboru, na který nemáte práva zápisu.
OutOfMemoryError	(výjimka nebyla uvedena)	Tuto chybu vrátí JVM, pokud zjistí, že již nelze programu přiřadit další paměť, ačkoliv by to bylo potřeba. Nejčastěji se stává, pokud omylem vytváříte objekty v nekonečném cyklu.

3.4 ZDROJE

[1] **Pavlíček, Luboš a Pavlíčková, Jarmila.** *Úvod do Javy.* Praha : Oeconomica, 2005. 80-245-0963-6.

[2] **Barnes, David J. a Kölling, Michael.** *Objects First With Java, A Practical Introduction Using BlueJ.* Second edition. Harlow : Pearson Education, Ltd., 2005. 0131-24933-9.

[3] **Pecinovský, Rudolf.** *Myslíme objektově v jazyku Java.* 2. aktualizované a rozšířené vydání. Praha : Grada Publishing a.s., 2009. 978-80-247-2653-3.

[4] **Sierra, Kathy a Bates, Bert.** *Head First Java.* místo neznámé : O'Reilly Media, 2005. 0-596-00920-8.

[5] **Pecinovský, Rudolf.** *OOP Naučte se myslet a programovat objektově.* Brno : Computer Press, a.s, 2010. 978-80-251-2126-9.

[6] **Horstman, Cay.** *Big Java.* New York : John Wiley & Sons, Inc., 2010. 0-471-40248-6.

[7] **Zakhour, Sharon a kol.** *Java 6 - Výukový kurz.* [překl.] Jakub Mikulaštík. Brno : Computer Press, 2007. 978-80-251-1575-6.

[8] **Pitner, Tomáš.** *Java - začínáme programovat.* Praha : Grada Publishing, s. r. o., 2002. 80-247-0295-9.

[9] **Kölling, Michael.** *Introduction to Programming with Greenfoot.* New Jersey : Pearson Education, Inc., 2010. 0-13-245428-9.

[10] **Winder, Russel a Roberts, Graham.** *Developing Java Software.* Third Edition. Chichester : John Wiley & Sons, Ltd., 2006. 0-470-09025-1.

[11] **Sun (Oracle).** The Java Tutorials. [Online] [Citace: 28. Duben 2010.] http://java.sun.com/docs/books/tutorial/.

[12] —. Java Platform SE 6 API Specification. [Online] [Citace: 28. Duben 2010.] http://java.sun.com/javase/6/docs/api/.

[13] **McConnel, Steve.** *Dokonalý kód - umění programování a techniky tvorby software.* [překl.] Bogdan Kiszka. Brno : Computer Press, a.s., 2006. 80-251-0849-X.
[14] compile time error messages : Java Glossary. *MindProd.* [Online] [Citace: 4. Duben 2010.] http://mindprod.com/jgloss/compileerrormessages.html.
[15] SQL Injection. *Wikipedia.* [Online] [Citace: 2. Květen 2010.] http://cs.wikipedia.org/wiki/SQL_injection.
[16] **Vysoká škola ekonomická v Praze.** Sylabus předmětu 4IT101 - Základy programování (FIS - LS 2009/2010). *Integrovaný studijní informační systém.* [Online] 12. prosinec 2009. [Citace: 1. květen 2010.] https://isis.vse.cz/katalog/syllabus.pl?predmet=63396.
[17] —. Sylabus předmětu 4IT115 - Základy softwarového inženýrství (FIS - LS 2009/2010). *Integrovaný studijní informační systém.* [Online] 14. prosinec 2009. [Citace: 1. květen 2010.] https://isis.vse.cz/katalog/syllabus.pl?predmet=63397.
[18] *Metodika Design patterns first - jak lze drobnou změnou výkladu zlepšit pochopení některých objektových konstrukcí.* **Pecinovský, Rudolf.**