Slezská univerzita v Opavě
Obchodně podnikatelská fakulta v Karviné

ZÁKLADY OBJEKTOVÉHO PROGRAMOVÁNÍ

Klepněte sem a zadejte text.

Pro prezenční formu studia

Roman Šperka, Jan Górecki

Karviná 2012

Projekt OP VK č. CZ.1.07/2.2.00/28.0017
„Inovace studijních programů na Slezské univerzitě,
Obchodně podnikatelské fakultě v Karviné“

Obor: Informatika.

Anotace: Studijní text je určen studentům bakalářského stupně na vysokých školách ekonomického zaměření. Je zaměřen na podporu výuky základů objektového programování. Obsahově je zaměřen na objektově orientované programování v jazyce Java. Text je doplněn o řadu praktických příkladů a ukázek zdrojového kódu. Tyto ukázky mohou čtenáři využít k tvorbě vlastních aplikací. V úvodních kapitolách je student obeznámen s filosofií a paradigmatem objektově orientovaného programování, architekturou Java, s instalací a popisem vývojového prostředí Eclipse Indigo. V dalších častech čtenář nalezne popis základních prvků jazyka Java a příkazů. Jádrem textu je popis základních konstruktů jazyka Java a to zejména metod, konstruktorů, dědičností a polymorfismem. Výklad končí popisem ladění programů a využitím balíčků při práci s jazykem. Poslední kapitola představuje praktický komplexní příklad implementace aplikace.

Klíčová slova: objektově orientovaný návrh, Java, programování, objekty, kód, implementace.

Autor: **Mgr. Ing. Roman Šperka, Ing. Jan Górecki**

Recenzenti: Doc. RNDr. Petr Šaloun, Ph.D., Mgr. Petr Suchánek, Ph.D.

ISBN Doplní oddělení vědy a výzkumu.

OBSAH

ÚVOD

Cílem předkládaného textu je seznámit čtenáře se základy objektově orientovaného programování. Objektově orientované programování (dále jen OOP) představuje v dnešní době rozšířenou metodu vytváření programů. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce. Základními programovacími prvky v OOP jsou útvary zvané objekty. Pro účely výkladu a objasnění hlavních pojmů OOP bude v učebním textu využito vývojového prostředí Eclipse Indigo a programovacího jazyka Java.

Současně s vývojem výpočetní techniky se vyvíjely i programovací jazyky, a to směrem k lepší srozumitelnosti pro programátory. Po dlouhou dobu byl zaběhnutý model strukturovaného (procedurálního) programováni, kde se program sestává z dat (uložených v proměnných) a procedur (funkcí), které s nimi pracují. Tento přístup dělil jednotlivé úkony do samostatných procedur, což bylo sice velice efektivní, ale postupem času při vytváření rozsáhlých a složitých aplikací se ukázaly i nevýhody tohoto modelu. Především ztráta přehlednosti kódu, obtížné rozšiřování aplikací a poměrně velká náchylnost k chybám. Kvůli těmto nedostatkům bylo vyvinuto OOP. Nový koncept OOP byl veřejnosti poprvé představen roku 1967, kdy Johan Dahl a Kristen Nygaard představili v Lisebu (město v Norsku poblíž Osla) první objektové orientovaný jazyk SIMULA 67. Významným programovacím jazykem, který výrazně ovlivnil moderní programování je Smalltalk 80, vyvinutý v PARC Xerox. OOP převzalo nejlepší myšlenky ze strukturovaného programování (řídící struktury apod.) a ty zkombinovalo s novými výkonnými koncepty, které dovolují organizovat programy mnohem efektivněji. V OOP se již nesnažíme program vnímat jako sadu proměnných a funkcí, ale naopak se snažíme co nejblíže přiblížit lidskému myšlení. OOP podněcuje k dekompozici problému na základní prvky. Každá komponenta se stává nezávislým objektem, který obsahuje vlastní instrukce a data. Tímto způsobem je komplikovanost snížena a programátor může pracovat s rozsáhlejšími programy.

Struktura učebního textu je nasledující. V úvodních částech je objasněná filosofie a paradigmata OOP, dále pak následuje výklad Java architektury. V těchto pasážích se čtenář obeznámí s Java technologiemi, kterými jsou např. servlety, java beany, dostupné frameworky apod. Další části se věnují instalaci a popisu prostředí Eclipse Indigo, základním prvkům a příkazům jazyka Java. Jádrem textu je představení Java konstruktů jako např. třída, objekt, metoda, konstruktor, destruktor apod. V druhé polovině textu je vysvětlen princip dědičnosti, polymorfismu a balíčků. Každá část je ilustrováná praktickými ukázkami zdrojového kódu, které umožňují zájemcům si je vyzkoušet. Poslední kapitola je věnovaná komplexní případové studii, která je shrnutím probírané problematiky.

1 ÚVOD DO OBJEKTOVÉHO PROGRAMOVÁNÍ

1.1 PARADIGMA OBJEKTOVÉHO PROGRAMOVÁNÍ

Strukturované programování nebo také strukturovaný programovací jazyk označuje programovací techniku, kdy se implementovaný algoritmus rozděluje na dílčí úlohy, které se spojují v jeden celek. **Objektově orientované programování** představuje v dnešní době nejrozšířenější metodu vytváření programů oproti klasické metodě strukturovaného programování. Metody OOP napodobují vzhled a chování objektu z reálného světa s možností velké abstrakce. Základními programovacími prvky v OOP jsou útvary zvané **objekty**. Tyto útvary v zásadě modelují objekty reálného světa: osoby, předměty, dokumenty, události. Každý objekt je nositelem jistých informací o sobě samém (tzv. **stavu**) a má schopnost na požádání tento stav měnit. Například objekt faktura z podnikového informačního systému má svoje číslo a platební informace. Svůj stav může změnit, například z neproplacené faktury se může stát faktura proplacená apod.

V objektově orientovaném paradigmatu je program strukturalizován nikoli podle procedur, které se vykonávají, ale podle objektů, které se v systému vyskytují. **Objekt** v sobě zapouzdřuje jak data, tak procedury pracující nad těmito daty. Průběh výpočtu je pak určen **posíláním zpráv** mezi jednotlivými objekty. Pomocí zpráv objektům říkáme, jak mají změnit svůj stav. Na zaslanou zprávu oslovený objekt nějak reaguje. Tuto reakci definujeme v kódu, který bývá označován jako **metoda**. Programátoři proto většinou neříkají, že objekt posílá druhému objektu zprávu, ale že volá jeho metodu. Činnost metody, tj. reakce osloveného objektu na zprávu, záleží na tom, kým a jak byla zpráva odeslána, a na tom, v jakém stavu se objekt v okamžiku obdržení zprávy nacházel.

Víme, že objekty se v okolním světě často opakují. **Třídu** můžeme chápat jako šablonu, která definuje, jak se budou vytvářet objekty, které označíme jako **instance** dané třídy. Tato šablona definuje, jaké budou mít její instance atributy a jaké budou mít metody.

Shodnost atributů jednotlivých instancí ale neznamená shodnost jejich hodnot. Můžeme např. definovat třídu Auto, jejíž instance budou mít atribut motor. Každá instance však bude mít svůj vlastní motor a tyto motory se navíc mohou vzájemně lišit výkonem, spotřebou, provedením apod. Třída ale může definovat i **vlastní atributy a metody** a ty pak její instance sdílejí. Když instance změní hodnotu některého svého atributu, ostatní instance se o tom nedozvědí. Pokud však změní hodnotu atributu třídy, budou to hned všechny instance vědět, protože s ní tento atribut sdílejí.

Při vývoji objektově orientovaných programů se klade velký důraz na to, aby jednotlivé části programu nemohly využívat své „znalosti“ o tom, jak je protější část naprogramována. U všech entit (objekty, třídy, metody, atd.) se proto rozlišují dvě charakteristiky:

- **Rozhraní**, které definuje, co o dané entitě ví okolní program.
- **Implementace**, která specifikuje, jak je dosažené správné funkčnosti dané entity.

Jedním z nejdůležitějších pravidel správného objektově orientovaného programování je zásada programovat proti rozhraní a ne proti implementaci. Moderní

programovací jazyky se snaží neponechávat dodržování této zásady pouze na programátorovi, ale snaží se její dodržení kontrolovat a případně také vynucovat.

Se skrýváním implementace souvisí nejdůležitější rys objektově orientovaného programování, kterým je **zapouzdření**. Zapouzdřením označujeme dvě věci:

- Umístění dat (atributů) a kódu (metod), který s těmito daty pracuje, pohromadě do definice třídy. Programátor tak má přehled o vlastnostech a stavu zpracovávaných dat a dělá proto mnohem méně chyb.
- Znemožnění ostatním částem programu manipulovat s těmito daty jakýmkoliv jiným způsobem, než prostřednictvím metod vlastníka těchto dat, tj. metod příslušného objektu. Jinými slovy: k datům objektu je možné přistupovat pouze způsobem definovaným v rozhraní daného objektu.

Důsledné zapouzdření všech částí programu dramaticky zvyšuje efektivitu vývoje i spolehlivost výsledného programu. Všechny další rysy jazyka jsou proto často posuzovány podle toho, nakolik podporují nebo naopak narušují optimální zapouzdření.

Jednou z konstrukcí, které nabízí většina objektových programovacích jazyků, je možnost definice dědičnosti. **Dědičnost** představuje speciální druh skládání, při němž je do objektu vložen jiný objekt, tzv. podobjekt předka, a nový majitel tohoto vloženého objektu přebírá jeho rozhraní a případně k němu přidává své vlastní rysy. Vložený objekt, resp. jeho třída, je pak označen za předka (odtud také název podobjekt předka) a jeho majitel za potomka. Dědičnost přináší elegantní způsob, jak si ušetřit programování. Potomci totiž dědí všechny dostupné metody svých předků, takže jim stačí, když sami definují pouze ty, jejichž definice předka jim nevyhovuje, a ty, které předek vůbec nemá. Dědičnost je však typickou konstrukcí, která narušuje zapouzdření, a to v obou jeho rysech:

- Potomkům bývá často umožněno pracovat přímo s daty svého předka, takže přestává platit, že kód je blízko zpracovávaných dat se všemi z toho plynoucími důsledky.
- Předek musí často prozradit potomkovi něco o své implementaci, protože jinak by potomek nebyl schopen definovat své metody bezchybně. To opět zvyšuje pravděpodobnost chyb.

Obecná zásada proto říká, že dědičnost máme použít pouze tehdy, když jakékoliv jiné řešení je výrazně těžkopádnější. Kromě toho platí další zásada: má-li program zůstat stabilní, musí být potomek vždy speciálním případem předka.

Jak jsme již uvedli v souvislosti se zapouzdřením, v objektově orientovaném programování se důsledně odděluje rozhraní a implementace. Programovací jazyk Java dokonce zavedl speciální konstrukci **interface**, která definuje pouze rozhraní bez jakékoliv implementace. Interface své metody pouze deklaruje, avšak nijak je neimplementuje. To ponechává na třídách, které se přihlásí k jeho implementaci. Takové třídy pak mohou své instance vydávat za instance implementovaného interfacu. Protože interface nedefinuje **žádnou implementaci**, nemůže mít ani **žádné instance**. Požaduje-li některá část programu instancí interfacu, musí ji zastoupit instance nějaké třídy, která daný interface implementuje.

Mohli bychom říci, že implementovaný interface se chová podobně jako předek – implementující třída také přebírá jeho rozhraní. Protože však instance implementující třídy nepřebírají žádný podobjekt předka, nemůže dojít ani k jednomu z obou výše uvedených problémů, s nimiž se setkáváme u dědičnosti.

Současné programování výrazně ovlivnil příchod **návrhových vzorů**[1]. Jejich používání zvyšuje efektivitu vývoje, spolehlivost a robustnost výsledných programů, jejich spravovatelnost a umožňuje mnohem rychlejší reakce na změny zadání.

Návrhové vzory bychom mohli označit za programátorskou verzi matematických vzorečků. Jenom se do nich nedosazují čísla, ale třídy, objekty a v některých případech metody. Znalost návrhových vzorů přináší několik výhod:

- Vývoj se zrychluje, protože programátoři nemusí v řadě případů vymýšlet vlastní dostatečně dokonalé a efektivní řešení.
- Díky tomu se vývoj i zkvalitňuje, protože odpadá možnost vzniku chyby při vývoji tohoto speciálního řešení.
- Vývoj se dále zlevňuje, protože části programu navržené s využitím návrhových vzorů je možné mnohem snadněji použít i v dalších programech.
- Zlepšuje se i komunikace mezi členy týmu. Když programátor řekne, že někde použil „jedináčka“ či „most“, všichni vědí, co od daného řešení mohou čekat a kde na ně číhají jaká úskalí a nemusí si vše vzájemně sáhodlouze vysvětlovat.

Zkušenost z 80. a zejména pak z 90. let ukázala, že nezávisle na tom, jak dokonalá bude analýza, program se bude s nejvyšší pravděpodobností měnit – jednou kvůli dodatečné změně v zadání, jindy kvůli novým technologiím. Byly proto vypracovány metody, jak program upravit, aby takovéto pozdní zásahy přišly co nejlevněji. Tyto techniky úpravy bývají označovány jako **refaktorizace** kódu (refactoring). V průběhu let se natolik osvědčily, že jsou nyní integrální součástí všech lepších vývojových prostředí.

Dalším velkým přínosem objektového programování je zavedení **automatizovaných jednotkových testů**[2]. Nové knihovny umožnily navrhovat testy mnohem efektivněji, než tomu bylo v minulém století, a tyto testy pak automaticky spouštět. Řada nových metodik dokonce prosazuje tzv. **programováním řízené testy**, které vyžaduje, aby programátor nejprve napsal testy vytvářeného programu, a teprve pak začal programovat. Jeho cíl se tím velmi zjednoduší (a tím stoupne jeho produktivita): jeho úkolem nyní totiž bude pouhé zprovoznění předem připravených testů. Podle této metodiky programátor spouští automatizované testy po každé, byť nepatrné změně programu. Úspěšné proběhnutí testů mu umožní se spoléhat na doposud napsaný kód a programovat v další etapě o to efektivněji. Naopak havárie testů oznámí, že chybu do programu zanesla některá z naposledy zanášených změn. Spouštíme-li jednotkové testy dostatečně často, bude těchto změn jenom velice málo a nemělo by dělat problém si pamatovat, co vše se zaměnilo, a snadno tak odhalit, kde je chyba.

Objektově orientované programování přineslo zásadní posun v kvalitě programování velkých systémů a umožnilo rychlejší vývoj programů.

[1] Návrhový vzor (anglicky design pattern) představuje obecné řešení problému, které se využívá při návrhu programů. Návrhový vzor není knihovnou nebo částí zdrojového kódu, která by se dala přímo vložit do našeho programu. Jedná se o popis řešení problému nebo šablonu, která může být použita v různých situacích.

[2] Jejich cílem je strojově otestovat, zda daná jednotka dělá to, co dělat má, a zda nedělá něco, co dělat nemá.

NEZAPOMEŇTE

Základní pojmy v OOP:

Objekty – jednotlivé prvky modelované reality (jak data, tak související funkčnost) jsou v programu seskupeny do entit, nazývaných objekty. Objekty si pamatují svůj stav a navenek poskytují operace (přístupné jako metody pro volání).

Abstrakce – programátor, potažmo program, který vytváří, může abstrahovat od některých detailů práce jednotlivých objektů. Každý objekt pracuje jako černá skříňka (black box), která dokáže provádět určené činnosti a komunikovat s okolím, aniž by vyžadovala znalost způsobu, kterým vnitřně pracuje.

Zapouzdření – zaručuje, že objekt nemůže přímo přistupovat k „vnitřnostem“ jiných objektů, což by mohlo vést k nekonzistenci. Každý objekt navenek zpřístupňuje **rozhraní**, pomocí kterého (a nijak jinak) se s objektem pracuje.

Třídu můžeme chápat jako šablonu, která definuje, jak se budou vytvářet objekty, které označíme jako instance dané třídy. Tato šablona definuje, jaké budou mít její instance atributy a jaké budou mít metody.

Skládání – Objekt může obsahovat jiné objekty. Tyto objekty tvoří jeho **atributy**. Hodnoty atributů definují stav objektu a ovlivňují jeho vlastnosti.

Delegování – Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

Dědičnost – objekty jsou organizovány stromovým způsobem, kdy objekty nějakého druhu mohou dědit z jiného druhu objektů, čímž přebírají jejich schopnosti, ke kterým pouze přidávají svoje vlastní rozšíření. Tato myšlenka se obvykle implementuje pomocí rozdělení objektů do tříd, přičemž každý objekt je instancí nějaké třídy. Každá třída pak může dědit od jiné třídy (v některých programovacích jazycích i z několika jiných tříd).

Polymorfismus – odkazovaný objekt se chová podle toho, jaké třídy je instancí. Pokud několik objektů poskytuje stejné rozhraní, pracuje se s nimi stejným způsobem, ale jejich konkrétní chování se liší podle implementace. U polymorfismu podmíněného dědičností to znamená, že na místo, kde je očekávána instance nějaké třídy, můžeme dosadit i instanci libovolné její podtřídy, neboť rozhraní třídy je podmnožinou rozhraní podtřídy. U polymorfismu nepodmíněného dědičností je dostačující, jestliže se rozhraní (nebo jejich požadované části) u různých tříd shodují, pak jsou vzájemně polymorfní.

1.2 JAVA TECHNOLOGIE

1.2.1 KOMPILERY A INTERPRETERY

Programovací jazyky můžeme rozdělit do dvou kategorií podle toho, jak jsou překládány a spouštěny. První skupinou jsou jazyky **interpretované** (např. PHP, Basic, JavaScript, atd.), druhou jsou pak kompilované jazyky (např. C, C++, C#). Interpretované jazyky se většinou provádí tak, že interpreter čte daný program

postupně a postupně ho také kontroluje, překládá (na volání nativních instrukcí procesoru) a provádí. Pokud narazí na chybu, potom vykonávání většinou přeruší a danou chybu oznámí. Interpretované jazyky mají menší nároky na formálnost kódu (netřeba inicializovat proměnné, datové typy se mohou za běhu měnit apod.). V zásadě jsou tyto jazyky pomalejší než kompilované, jelikož interpretace a překlad do nativních instrukcí trvá nějaký čas. Navíc se tyto programy musí vždy spouštět v interpreteru. **Kompilované** programy jsou celé přeloženy do instrukcí cílového procesoru a až potom mohou být spušněny (procesor vykonává dané instrukce). Jsou rychlejší, mají vyšší nároky na formální správnost kódu. Překádají se kompilátorem, výsledkem překladu je (většinou ve Windows) .exe soubor.

1.2.2 HISTORIE JAZYKA JAVA

Historie jazyka Java sahá daleko do doby před uvedením první ostré verze firmou Sun Microsystems. Až od roku 1995 je ale skutečně nazývána Javou (předtím Oak). Traduje se, že tento název vznikl ze slangového označení kávy, kterou má Java také ve svém logu. Za několik málo let své existence prošla Java bouřlivým vývojem, byly vytvořeny tři “poddruhy” pro mobilní telefony (ME), desktopy (SE) a distribuované systémy (EE). Díky nim už jazyk Java není ani tak programovacím jazykem, jako spíše celou platformou. Velké popularity se jazyk dočkal výhradně díky své skvělé přenositelnosti a dodnes je extrémně používaným. Zatímco použití na desktopech mírně stagnuje, mobilní ooblast jazyka zažívá už několik let stabilní rozkvět. V roce 2007 Sun navíc uvolnil všechny zdrojové kódy Javy (2,5 milionu řádků) a tak je od této doby Java vyvíjena jako opensource, co prospělo její masovému použití.

1.2.3 JAVA PROSTŘEDÍ

Java leží někde na pomezí, dá se říci, že je obojí, nejdříve je totiž zdrojový java soubor kompilován do tzv. bajtového kódu a ten je následně interpretován virtuálním strojem (tedy ne přímo procesorem daného počítače!). Díky skutečnosti, že je bajtový kód (bytecode) interpretován virtuálním strojem, je java považována spíše za interpretovaný jazyk. Java (program a prostředí nutné pro běh) se skládá ze 4 součástí, jedná se o:

- Programovací jazyk Java.
- Class soubory (kompilované java soubory se zdrojovým kódem).
- API (programové rozhraní).
- Virtuální stroj Javy.

Virtuální stroj spolu s **JRE** (Java Runtime Environment), který je jeho součástí převádí bytecode do nativních instrukcí procesoru. JRE je prostředí pro zpracování jazyka Java. Obsahuje třídy výkonného jádra Javy a již zmíněný virtuální stroj. Jazyk Java se skládá z několika konfigurací, podle toho, k jakému účelu slouží, jedná se o konfigurace:

- **Java ME** – mikro edice Javy. Obsahuje třídy a konfigurace pro mobilní a přenosná zařízení, navigace, pagery apod.
- **Java SE** – standardní edice Javy. Obsahuje základní sadu tříd (jádro, práce se vstupy, výstupy, atd.). Disponuje také sadou pro tvorbu grafického uživatelského rozhraní (UI – User Interface).

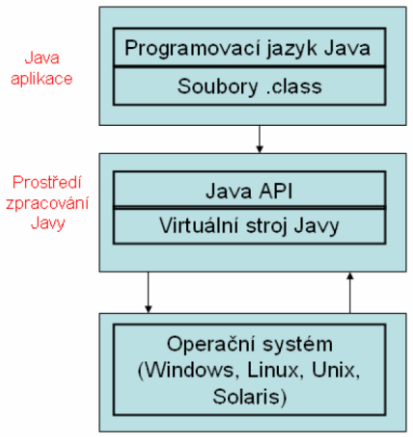
- **Java EE** – enterprise edition. Obsahuje sadu rozhraní a tříd především pro vývoj podnikových a webových aplikací (servlety, JSP, EJB apod.).

Pokud však mluvíme o jazyku Java, jeho konfiguracích a použití, musíme zmínit také **Java Community Process** (JCP). Jedná se o formalizovaný proces navrhování a schvalování budoucích verzí a rysů Javy. Ustanoven byl roku 1998. Hlavním výstupem procesu JCP je požadavek na specifikace zvaný **JSR** (Java Specification Request):

- Jedná se o formální dokument navrhující specifikace a technologie, které by měly být přidány do Javy.
- Probíhá formální veřejné review dříve, než se JSR stane finálním.
- JSR poskytuje referenční implementaci.

V dnešní době existuje něco kolem 300 JSR.

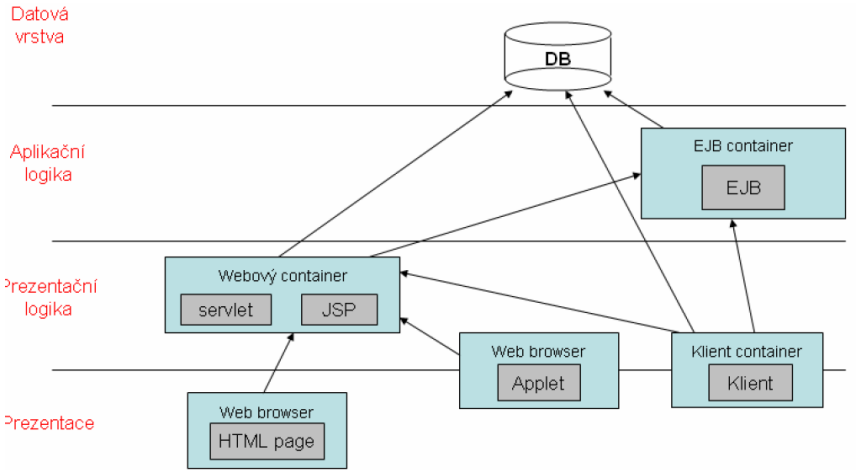
Obrázek 1-1: Prostředí pro zpracování jazyka Java.



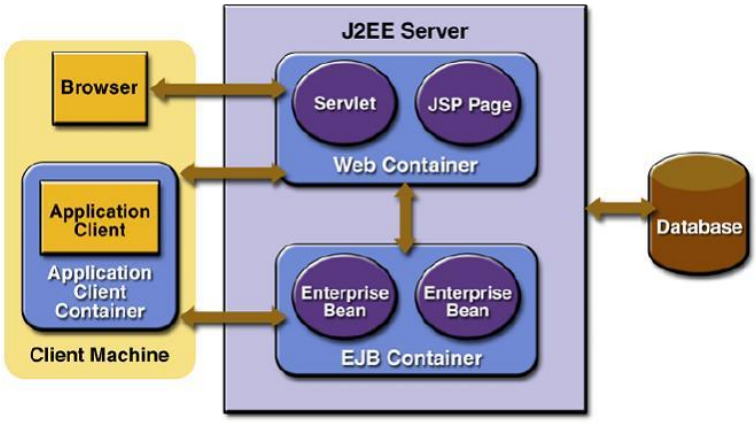
1.3 JAVA ENTERPRISE EDITION

Java má použití (hlavně) v podnikových aplikacích, kde je důležitým kritériem bezpečnost, škálovatelnost, výkonnost či přenositelnost dané aplikace. API, které poskytuje standardní Java je samozřejmě možné použít i pro podnikovou Javu. Následující obrázky ukazují některé specifikace Java EE a jejich použití v podnikové architektuře:

Obrázek 1-2: Java EE komponenty v logických vrstvách aplikací.



Obrázek 1-3: J2EE server a kontainery.



2 INSTALACE A POPIS PROSTŘEDÍ

Předmět Základy objektového programování obsahuje kromě teoretické části také část praktickou, ve které si budeme jednotlivé teoretické koncepty ilustrovat na jednoduchých příkladech, které jsou implementovány v objektově orientovaném programovacím jazyce Java. Abychom mohli programy v jazyce Java jednoduše vytvářet, upravovat a spouštět, je potřeba nainstalovat si k tomu vhodný software. Odpověď na otázku „Jaký software si nainstalovat a jak s ním pracovat?“ bude náplní následující části této kapitoly.

2.1 JAK NAINSTALOVAT JAVU

Napsat jednoduchý program v Javě není nic složitého. Aby bylo možné takový program spustit, je nutné nejprve nainstalovat překladač a interpret, který překlad a spuštění napsaného programu umožní. Instalace těchto komponent (souhrnně označovaných jako JDK) není náročná a probíhá v podstatě jako instalace jakéhokoliv programu. Tak jako všechny programy i JDK je průběžně aktualizováno nejrůznějšími updaty. Instalace je však vždy stejná, liší se pouze číslo aktualizace. V současné době je k dispozici instalace JDK 7.0 Update 7.

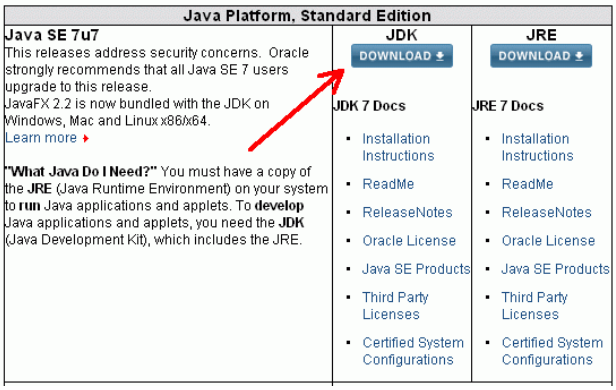
Pokud budeme postupovat podle následujícího podrobného návodu, bude pro nás nainstalování Javy a nastavení všech potřebných cest hračkou. Musíme provést čtyři základní kroky:

1. Stažení JDK.
2. Instalace Javy.
3. Správné nastavení cest, aby vše fungovalo.
4. Instalace dokumentace.

2.2 STAŽENÍ JDK

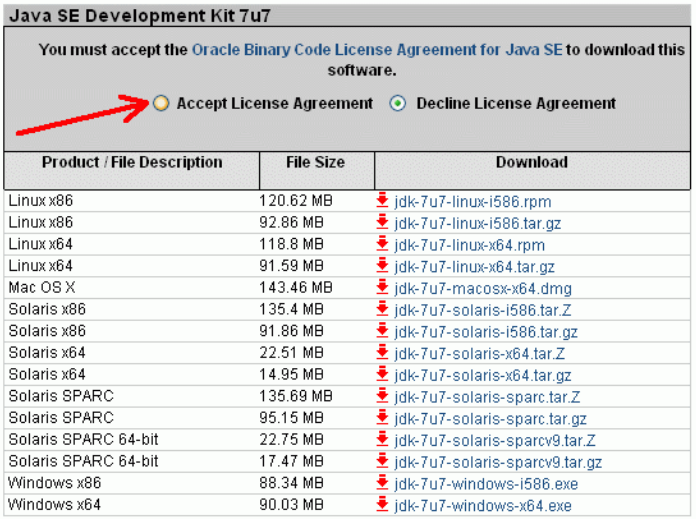
Samotné JDK je možné bezplatně stáhnout z oficiálních stránek Oracle http://www.oracle.com/technetwork/java/javase/downloads/index.html. (Na následujících 3 obrázcích jsou ukázány vždy výřezy www stránek, které jsou pro stažení souboru podstatné.)

Obrázek 2-1: Stránky pro stažení JDK



Po volbě Download musíme nejdříve potvrdit souhlas s licenčními podmínkami (Accept Licence Agreement)…

Obrázek 2-2: Přijetí podmínek pro stažení JDK



Java SE Development Kit 7u7

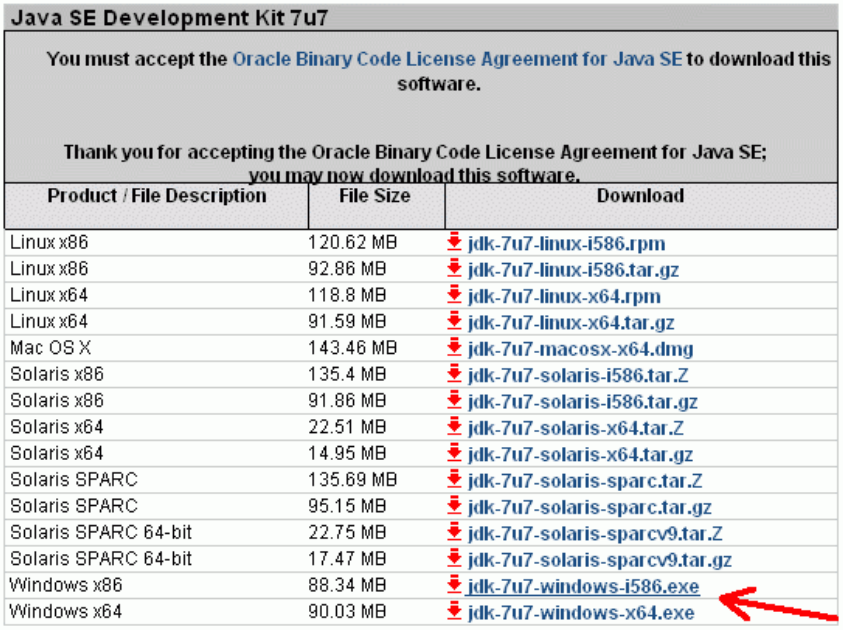
You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Accept License Agreement Decline License Agreement

Product / File Description	File Size	Download
Linux x86	120.62 MB	jdk-7u7-linux-i586.rpm
Linux x86	92.86 MB	jdk-7u7-linux-i586.tar.gz
Linux x64	118.8 MB	jdk-7u7-linux-x64.rpm
Linux x64	91.59 MB	jdk-7u7-linux-x64.tar.gz
Mac OS X	143.46 MB	jdk-7u7-macosx-x64.dmg
Solaris x86	135.4 MB	jdk-7u7-solaris-i586.tar.Z
Solaris x86	91.86 MB	jdk-7u7-solaris-i586.tar.gz
Solaris x64	22.51 MB	jdk-7u7-solaris-x64.tar.Z
Solaris x64	14.95 MB	jdk-7u7-solaris-x64.tar.gz
Solaris SPARC	135.69 MB	jdk-7u7-solaris-sparc.tar.Z
Solaris SPARC	95.15 MB	jdk-7u7-solaris-sparc.tar.gz
Solaris SPARC 64-bit	22.75 MB	jdk-7u7-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.47 MB	jdk-7u7-solaris-sparcv9.tar.gz
Windows x86	88.34 MB	jdk-7u7-windows-i586.exe
Windows x64	90.03 MB	jdk-7u7-windows-x64.exe

...a teprve poté je možné stáhnout samotný instalační soubor jdk-7u7-windows-i586.exe pro 32bitové windows nebo pro libovolnou jinou alternativu (v závislosti na verzi operačního systému, který používáme).

Obrázek 2-3: Stažení instalačního souboru



Java SE Development Kit 7u7

You must accept the Oracle Binary Code License Agreement for Java SE to download this software.

Thank you for accepting the Oracle Binary Code License Agreement for Java SE; you may now download this software.

Product / File Description	File Size	Download
Linux x86	120.62 MB	jdk-7u7-linux-i586.rpm
Linux x86	92.86 MB	jdk-7u7-linux-i586.tar.gz
Linux x64	118.8 MB	jdk-7u7-linux-x64.rpm
Linux x64	91.59 MB	jdk-7u7-linux-x64.tar.gz
Mac OS X	143.46 MB	jdk-7u7-macosx-x64.dmg
Solaris x86	135.4 MB	jdk-7u7-solaris-i586.tar.Z
Solaris x86	91.86 MB	jdk-7u7-solaris-i586.tar.gz
Solaris x64	22.51 MB	jdk-7u7-solaris-x64.tar.Z
Solaris x64	14.95 MB	jdk-7u7-solaris-x64.tar.gz
Solaris SPARC	135.69 MB	jdk-7u7-solaris-sparc.tar.Z
Solaris SPARC	95.15 MB	jdk-7u7-solaris-sparc.tar.gz
Solaris SPARC 64-bit	22.75 MB	jdk-7u7-solaris-sparcv9.tar.Z
Solaris SPARC 64-bit	17.47 MB	jdk-7u7-solaris-sparcv9.tar.gz
Windows x86	88.34 MB	jdk-7u7-windows-i586.exe
Windows x64	90.03 MB	jdk-7u7-windows-x64.exe

2.3 INSTALACE JAVY A SPRÁVNÉ NASTAVENÍ CEST

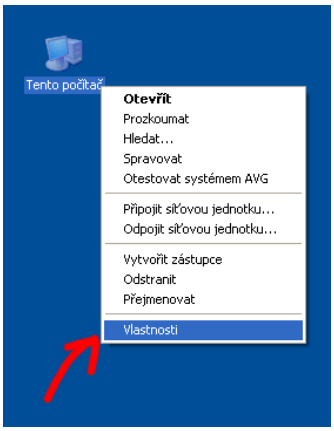
V tuto chvíli tedy máme k dispozici samotný instalační soubor. Spustíme soubor jdk-7u7-windows-i586.exe (z Průzkumníka nebo z Total Commanderu) a zahájíme vlastní instalaci. Po spuštění nás dialogové okno upozorní, že chceme instalovat Javu a informuje nás o instalované verzi apod., instalaci zahájíme stiskem tlačítka *Next*.

V dalším dialogovém okně můžeme volit komponenty, které chceme nainstalovat, a také měnit adresář, do kterého se Java nainstaluje (doporučené je nechat vše tak, jak je nastaveno implicitně). Během instalace nám Java nabídne těchto dialogů několik. Tato nastavení pro nás jako pro začátečníky nejsou nijak důležitá, takže necháme vše nastavené tak, jak je, a budeme v instalaci pokračovat stiskem tlačítka *Next*. Nakonec necháme Javu, ať se pěkně instaluje sama až do chvíle, kdy se instalace dokončí. Případná další dialogová okna potvrdíme tlačítkem *Next*. Závěrečné dialogové okno zavřeme stiskem tlačítka *Close*.

Teď máme Javu nainstalovanou, ale pokud bychom chtěli spustit nějaký napsaný program, ještě by se nám to s nejvyšší pravděpodobností nepodařilo. Aby všechno fungovalo tak, jak má, musíme ještě správně nastavit cesty k souborům překladače a interpreta.

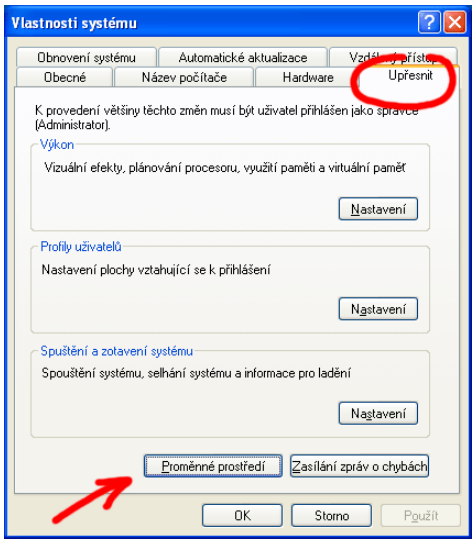
Kliknutím pravým tlačítkem na položce *Tento počítač* umístěné na pracovní ploše nebo v nabídce *Start* se nám v blízkosti kurzoru rozbalí kontextové menu. V něm vybereme položku *Vlastnosti*.

Obrázek 2-4: Vlastnosti tohoto počítače

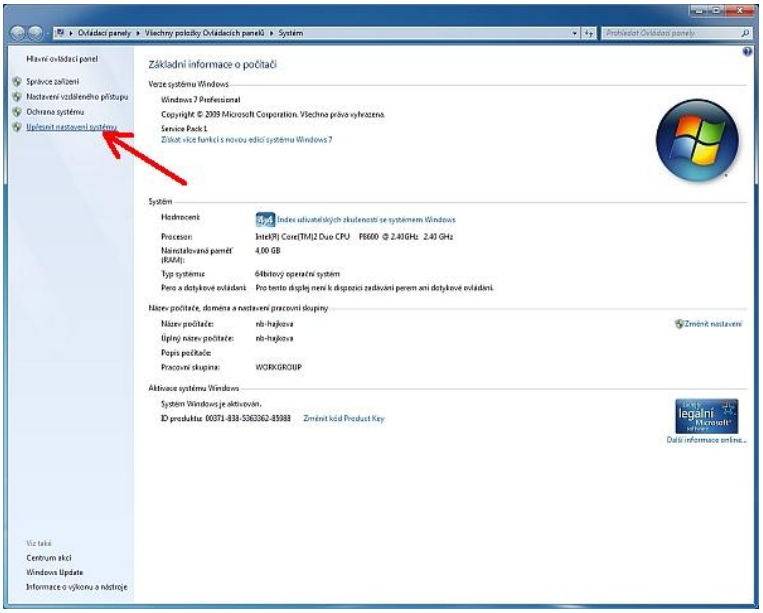


Máte-li Windows XP, jeho volbou se zobrazí dialogové okno s několika záložkami. Na záložce Upřesnit je umístěné tlačítko *Proměnné prostředí*. Pro nastavení cesty ho musíme stisknout. Pro Windows 7 Je třeba zvolit položku *Upřesnit nastavení systému*.

Obrázek 2-5: Nastavení cest pro Windows XP

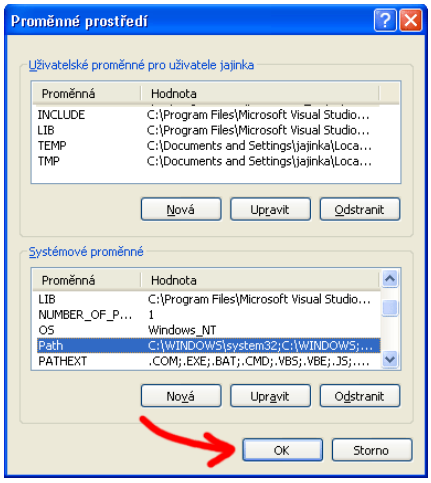


Obrázek 2-6: Nastavení cest pro Windows Vista a Windows 7



Poté se zobrazí další dialogové okno, které umožňuje nastavení uživatelských a systémových proměnných. Nás teď zajímá proměnná *Path* (= cesta). Ta je umístěná mezi systémovými proměnnými (tlačítko *Proměnné prostředí…*) ve spodní části dialogu. Označením proměnné *Path* a stiskem tlačítka *Upravit* (popř. přímo dvojklikem na proměnné *Path*) lze hodnotu této proměnné upravovat. U různých verzí operačního systému Windows se mohou tyto kroky lišit, důležité však je najít mezi proměnnými prostředí alternativu proměnné *Path* (v horní nebo spodní části okna) a tu upravit.

Obrázek 2-7: Dialogové okno Proměnné prostředí



K tomu je určené další dialogové okno obsahující název a hodnotu proměnné. V kolonce pro hodnotu proměnné je již uvedena řada cest pro ostatní aplikace. Tyto cesty tam tedy musíme ponechat nezměněné, jinak by ostatní aplikace mohly přestat (a pravděpodobně také přestaly) fungovat. Vše, co budeme do kolonky doplňovat, tedy doplníme až na konec. Za poslední znak připíšeme středník, tím oddělíme další přidávanou cestu, a za středník vložíme cestu v adresářové struktuře, v níž je

umístěn soubor java.exe. V našem případě tedy do kolonky *Hodnota* proměnné přidáme:

;C:\Program Files\Java\jdk1.7.0_07\bin;.;

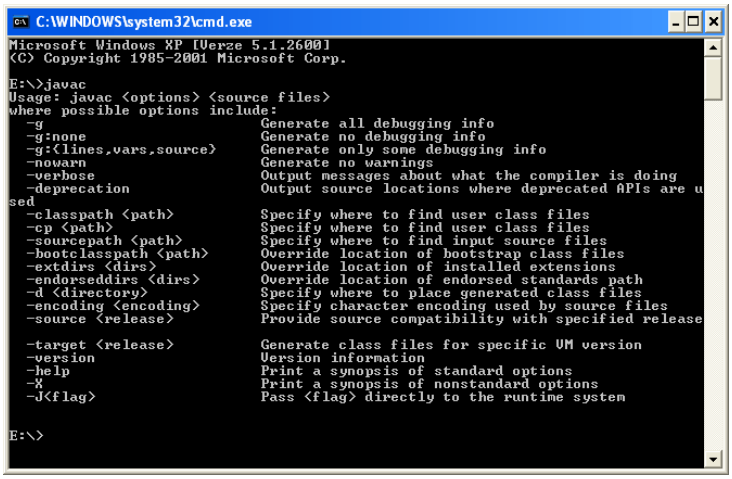
a stiskem tlačítka *OK* změnu potvrdíme. Dialogové okno se zavře a provedená změna se uloží do systému. Ve všech doposud otevřených dialogových oknech potvrdíme tlačítko *OK*.

Poznámka: Může se stát, že nemáte přístupová práva nastavena pro měnění systémových proměnných. V tomto případě upravte proměnnou *Path* v horní části dialogového okna (Uživatelské proměnné). Pokud tam proměnná *Path* ještě není, založte novou.

V tuto chvíli je vše nastaveno tak, jak má být a programy, které jsme napsali v Javě by měly jít bez problému překládat a spouštět. Pro jistotu provedeme restart počítače.

Poznámka: to, že vše funguje správně, zjistíme, když do příkazové řádky zadáme příkaz *javac* a systém vypíše nápovědu.

Obrázek 2-8: Nápověda vypsaná po spuštění příkazu javac při správném nastavení cest v systému



I přesto, že jsme všechno nastavili správně, může se nám časem stát, že nám Java přestane fungovat. Při instalaci některých programů je totiž možné, že změní proměnnou *Path* a odstraní důležitý středník a tečku ;.. Pokud nám tedy Java z ničeho nic přestane fungovat, je praktické proměnnou zkontrolovat a v případě potřeby tam středník a tečku znovu přidat a restartovat počítač.

2.4 INSTALACE VÝVOJOVÉHO NÁSTROJE ECLIPSE INDIGO

V předchozích třech podkapitolách jsme si ukázali, jak nainstalovat JDK. Nyní můžeme pro psaní programů v Javě použít např. libovolný textový editor, který neukládá do výsledného souboru navíc žádné informace pro formátování (např. MS Word je pro tento účel naprosto nepoužitelný, protože do výsledného souboru ukládá spoustu nadbytečných informací, se kterými by si překladač při spouštění programu neporadil).

Obyčejný textový editor však nezvýrazňuje syntaxi zdrojového kódu (nijak nám psaní programu nezpřehledňuje). Existují proto "chytřejší" editory, které během psaní

zdrojového kódu barevně odlišují klíčová slova, textové řetězce, číslice apod., takže je pro nás editovaný zdrojový text výrazně čitelnější (např. SciTe).

Půjdeme-li ještě dál a budeme od editoru požadovat další funkce, které nám programování usnadní (doplňování zdrojového kódu, debugger, snažší správu souborů apod.), najdeme celou řadu freewarových i placených produktů. Mezi nejpopulárnější a neustále se rozvíjející patří prostředí Eclipse Indigo. Tento nástroj, který mimo jiné nabízí příjemné pracovní prostředí, debugger a nemalé množství rozšiřujících pluginů, je možné bezplatně stáhnout z domovské stránky http://www.eclipse.org. V současné době je k dispozici jeho nejnovější verze Eclipse Indigo Sr2. Samozřejmě i používání prostředí Eclipse musí předcházet instalace Javy (JDK nejlépe 7.0).

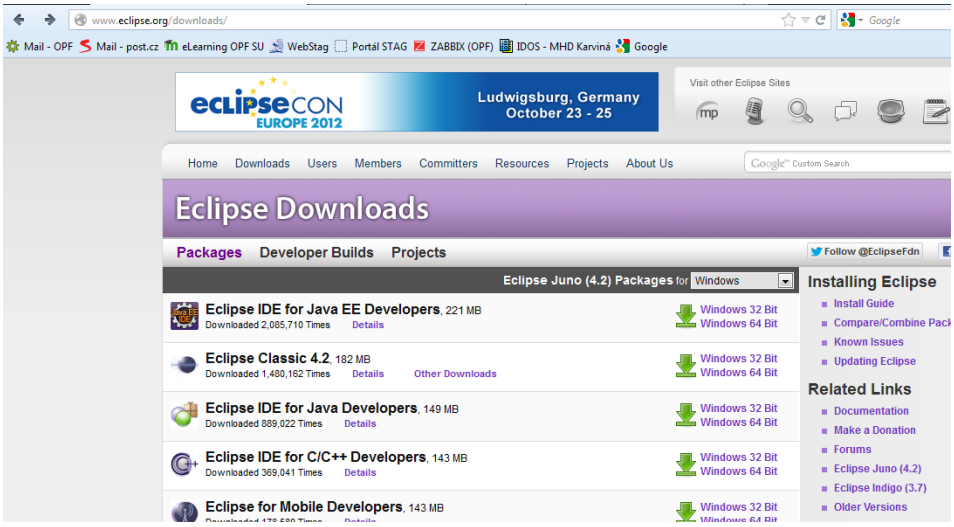
V následujícím textu si ukážeme, jak si Eclipse stáhnout, nainstalovat a jak ho jednoduše používat. Potřebné činnosti můžeme rozdělit do následujících několika kroků:

1. Stažení a instalace Eclipse SDK
2. První spuštění
3. Vytvoření projektu a souborů
4. Spuštění programu
5. Jak spustit samostatný zdrojový kód

2.5 STAŽENÍ A INSTALACE ECLIPSE INDIGO SR2

Abychom mohli Eclipse používat, musíme mít k dispozici soubor eclipse-jee-indigo-SR2-win32.zip (případně alternativní verzi pro jinou verzi operačního systému). Ten si můžeme stáhnout ze stránek Eclipse (viz. **Obrázek 2-9**), kde po pravé straně je pouze třeba si zvolit správnou verzi v závislosti na našem operačním systému.

Obrázek 2-9: Umístění Eclipse Indigo na stránkách eclipse.org

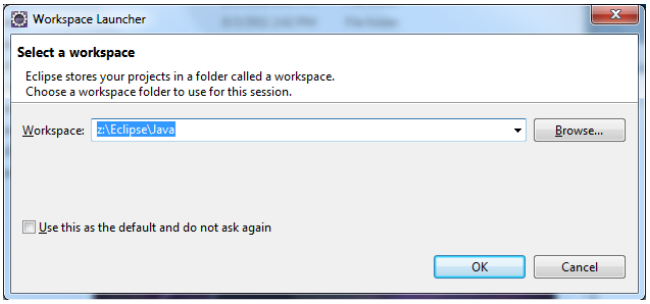


Předpokládejme, že jsme si instalační soubor opatřili. Co nás čeká teď? Soubor eclipse-jee-indigo-SR2-win32.zip je nutné rozbalit někam, odkud bude možné později program spouštět (Eclipse se neinstaluje, ale pouze rozbalí). Doporučené místo je C:\Program Files\eclipse. Adresář obsahuje několik složek a samostatných souborů. Mimo jiné je v něm umístěn také soubor eclipse.exe. Tento soubor slouží ke spuštění programu.

2.6 PRVNÍ SPUŠTĚNÍ

Po spuštění programu (spuštění souboru eclipse.exe) musíme ještě nastavit cestu k tzv. workspace (složce, kterou bude Eclipse používat jako pracovní **- Obrázek 2-10**). Tuto složku můžeme ponechat tak, jak se nastaví automaticky nebo si zvolit libovolnou jinou složku. Pokud nechceme, aby se tento dialog otevíral při každém spuštění, zaškrtneme volbu "Use this as the default and do not ask again". Je dobré si cestu k workspace pamatovat, abychom veděli, kde máme své soubory později hledat, pokud je potřebujeme například nakopírovat na flash disk a přenést je na jiný počítač. V případě absence instalace JDK, nebo jeho špatného nastavení nahlásí Eclipse chybu a nespustí se.

Obrázek 2-10: Volba umístění složky Workspace

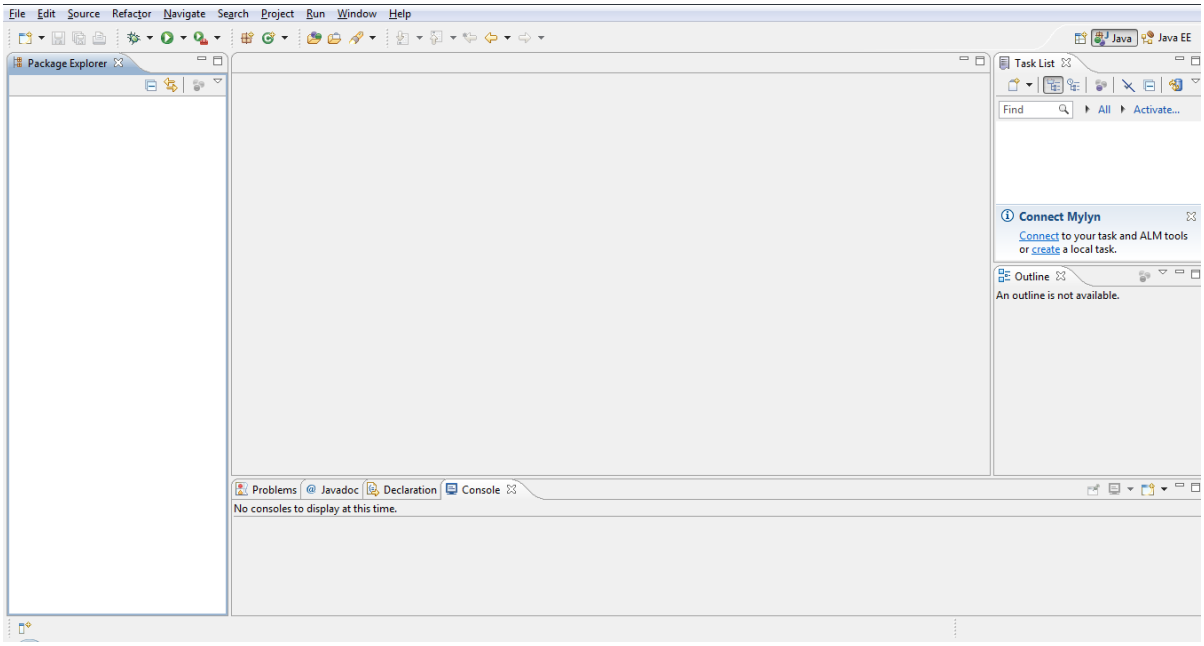


Poté se již spustí prostředí Eclipse s hlavní uvítací stránkou (záložka *Welcome*), kterou prozatím můžeme zavřít. Po jejím uzavření se nám pracovní prostředí otevře ve své plné kráse (viz. **Obrázek 2-11**). Co znamenají a k čemu slouží jeho jednotlivé části se dozvíme v dalších kapitolách.

2.7 VYTVOŘENÍ PROJEKTU A SOUBORŮ

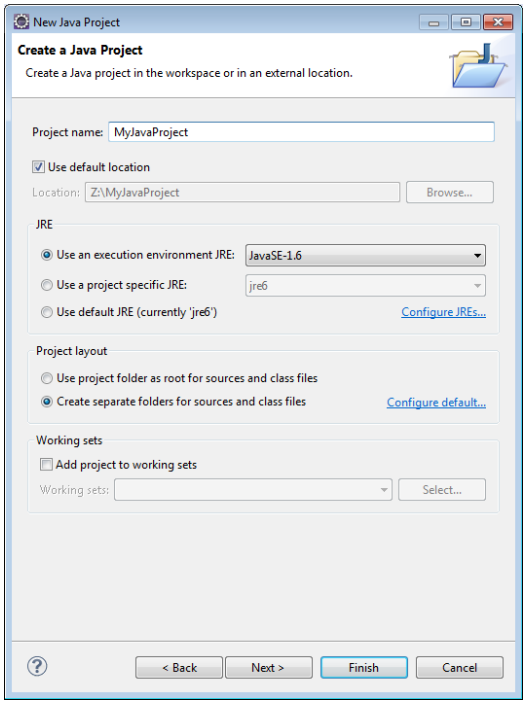
Zatím máme pracovní prostředí prázdné, pojďme se tedy podívat na to, jak vytvořit projekt, ve kterém budeme moci vytvářet soubory se zdrojovými kódy. Bez projektů se v Eclipsu neobejdeme, každý program musí být totiž uložen v některém z projektů, aby ho bylo možné spustit.

Obrázek 2-11: Prostředí Eclipse Indigo



Volbou *File -> New -> Project* a pak *Java Project* v menu získáme dialogové okno zobrazené na **Obrázek 2-12.**

Obrázek 2-12: Dialogové okno pro založení nového projektu

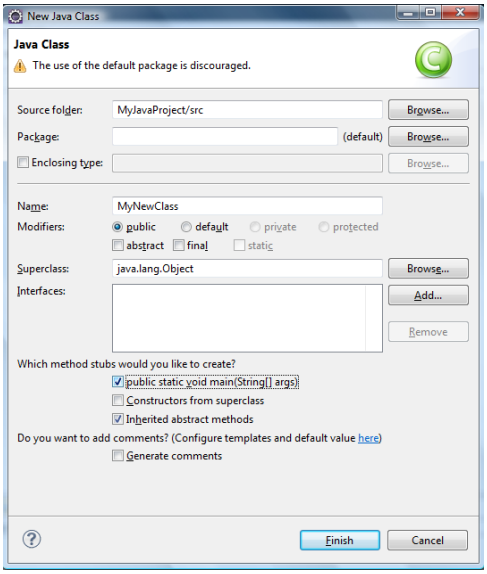


Zde můžeme nastavit několik vlastností zakládaného projektu. Pro začátek nám však bude stačit zadání jména projektu (*Project name*) a následný stisk tlačítka *Finish*. Poté se nám ještě zobrazí dialogové okno, které se nás zeptá, zda chceme námi založený projekt otevřít v tzv. přidružené perspektivě (o tomto dále v textu) – stiskneme tlačítko *Yes*, dialogové okno se zavře a v levém sloupci pracovního prostředí (tzv. *Package Explorer*) nám přibude první položka - námi založený projekt.

2.8 VYTVOŘENÍ NOVÉHO SOUBORU V PROJEKTU

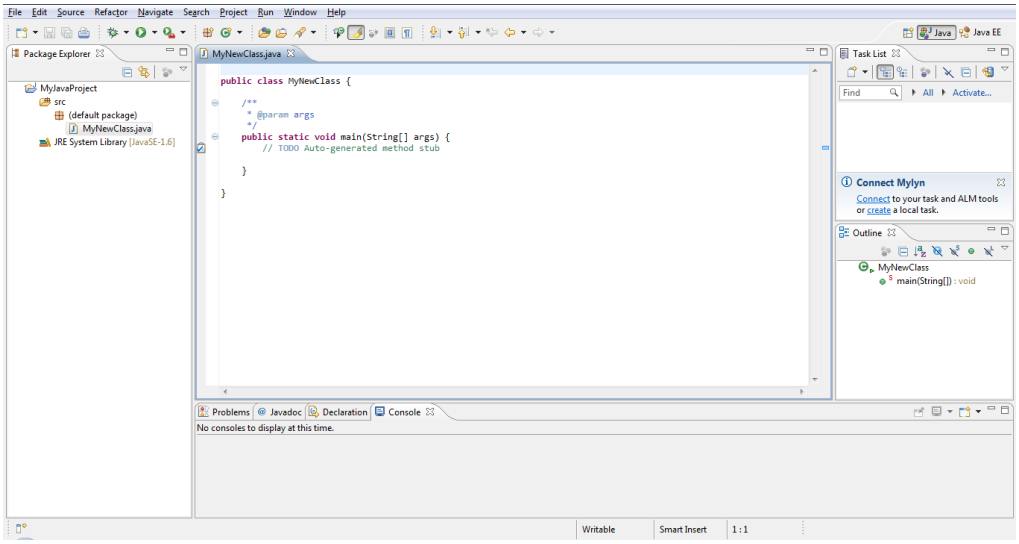
Vytvořili jsme projekt. Ten je však zatím prázdný, neobsahuje žádný soubor se zdrojovým kódem. Aby tedy celá naše snaha měla nějaký efekt (= abychom mohli spustit program napsaný v Javě), musíme teď nějaký program vytvořit. Vytvoříme tedy novou třídu - volbou *File -> New -> Class*, popř. tlačítkem v hlavní nástrojové liště. V zobrazeném dialogu můžeme zadávat parametry nově vytvořené třídě. Zadáme jméno třídy (*Name*), popř. také jméno balíku *Package*. Pokud necháme jméno balíku nevyplněné, je třída vložena do defaultního balíku. Abychom mohli nově vytvářený program spustit, zaškrtneme ještě políčko public static void main(String[] args) ve spodní části dialogového okna (viz. **Obrázek 2-13**), které nám zajistí doplnění šablony spustitelné metody main do nově vytvářené třídy. Na závěr stiskneme tlačítko *Finish*. Pokud bychom chtěli třídu později vložit do jiného balíku, můžeme ji pomocí myši jednoduše přetáhnout v *Package Exploreru*.

Obrázek 2-13: Dialogové okno pro tvorbu nové třídy



V *Package Exploreru* na levé straně obrazovky se nám teď v projektu objevil nově vytvořený soubor. Ten se zároveň otevřel v hlavní části okna a my ho můžeme libovolně editovat, ukládat a vytvořit tak takový zdrojový kód, jaký potřebujeme pro správnou funkčnost našeho programu (viz. **Obrázek 2-14**).

Obrázek 2-14: Vzhled prostředí po vytvoření nové třídy



2.9 PŘIDÁNÍ EXISTUJÍCÍHO SOUBORU DO PROJEKTU

Samozřejmě, že vytvoření nového souboru není jediná možnost, jak do projektu přidat soubor. Máme-li již zdrojový kód hotový (řekněme v souboru Soubor.java), stačí tento soubor nakopírovat do adresáře zdrojových kódů příslušného projektu v pracovní složce. Aby se změna projevila v *Package Exploreru*, klikneme na projekt, do kterého jsme soubory přidávali, a stiskneme klávesu F5 (*refresh*). Pozor, pokud používáme balíky, musí být i při kopírování zachována správná adresářová struktura, která balíku odpovídá.

Pokud se nám nechce kopírovat soubory mezi adresáři, můžeme využít ještě jiný způsob přidání existujících souborů do projektu. Jednoduše si soubory v jakémkoli správci souborů (Total Commander, Průzkumník, ...) nakopírujeme do schránky (CTRL+C, CTRL+Insert, nebo kliknutím pravým tlačítkem na označené soubory a volbou *Kopírovat* v menu). Poté se přepneme zpět do prostředí Eclipse, pravým tlačítkem myši klikneme na balík, do kterého chceme soubor přidat, a v menu zvolíme *Paste*, popřípadě označíme název balíku a stiskneme kombinaci kláves CTRL+V, popř. Shift+Insert. Soubory se tak vloží do projektu a jsou automaticky nakopírovány do adresářové struktury workspace.

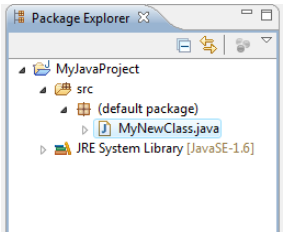
2.10 PRÁCE S PROSTŘEDÍM ECLIPSE

Tím, že jsme začali vytvářet zdrojový kód, začínají se nám objevovat položky i v ostatních částech pracovní plochy. Je tedy na čase si vysvětlit jejich základní význam.

2.10.1 LEVÝ SLOUPEC

Package Explorer - slouží pro správu projektů, jejich souborů, knihoven apod. Projekty uložené v pracovním prostředí lze otevírat a zavírat, u otevřeného projektu můžeme procházet jednotlivé soubory, seznamy lze rozbalovat a následně zase sbalovat. Kliknutím na soubor se daný soubor otevře v pracovní části okna.

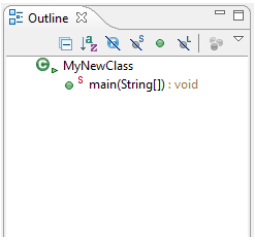
Obrázek 2-15: Package explorer v levé části pracovní plochy



2.10.2 PRAVÝ SLOUPEC

Outline - zobrazuje všechny proměnné, metody a třídy v právě otevřeném souboru. Kliknutím na příslušnou položku se kurzor v souboru přenese na její deklaraci.

Obrázek 2-16: Outline v pravé části pracovní plochy

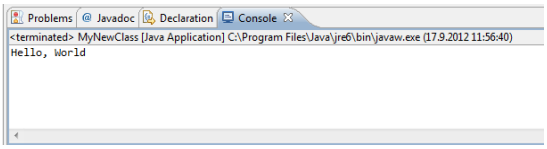


2.10.3 SPODNÍ ŘÁDKA

Obsahuje několik záložek, ze kterých nás budou zatím zajímat pouze dvě:

1. *Problems* - odkazuje na chyby (Errors) a upozornění (Warnings), poklikáním na hlášení chyby/warningu je daný problém vyznačen přímo ve zdrojovém kódu.
2. *Console* – zajišťuje textový výstup.

Obrázek 2-17: Výstup z konzole ve spodní části pracovní plochy



2.10.4 KLÁVESOVÉ ZKRATKY

Pro zjednodušení a hlavně programování nabízí Eclipse řadu funkcí pro práci se zdrojovým kódem (viz. **Tabulka 1**).

Tabulka 1: Klávesové zkratky

automatické doplňování	CTRL + mezerník
automatická korekce importů	CTRL + Shift + O
podtrhávání chyb	automaticky (aktualizace po uložení souboru - CTRL + S)
hromadné přejmenování názvů	ALT + Shift + R

rekompilace + spuštění	F11

2.11 „HELLO, WORLD!“ A JEHO SPUŠTĚNÍ

Téměř každý materiál sloužící k výuce programovacího jazyka začíná jednoduchým programem, který po spuštění vypíše hlášku „Hello, World!“. I zde tomu nebude jinak.

Následující příklad nejdříve nakopírujeme místo již vytvořené prázdné třídy MyNewClass.

PŘÍKLAD 1

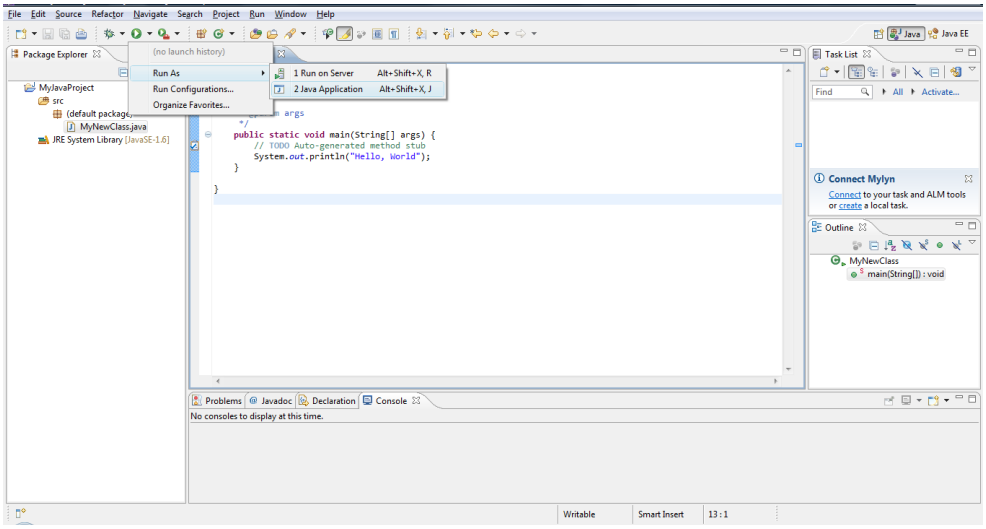
```
public class MyNewClass
{
  public static void main(String args[])
  {
     System.out.println("Hello, World!");
  }
}
```

Díky tomu, že třída MyNewClass obsahuje spustitelnou medodu main(), můžeme program spustit. Postup je následující: V menu zvolíme *Run -> Run As -> Java Application*. Pokud jsme po poslední úpravě soubor neuložili (CTRL + S), zobrazí se nám hláška, která nás k tomu vybídne. Poté, co je soubor uložen, program se spustí a dle očekávání vypíše hlášku Hello, World!.

Stejného efektu docílíme kliknutím na ikonku a volbami *Run As -> Java Application* (viz. **Obrázek 18**). V obou případech musíme mít v hlavním okně otevřený soubor obsahující metodu main(), ve kterém je aktivní kurzor. Další možností, jak spustit program, je kliknout na název souboru obsahující metodu main() v *Package Exploreru* pravým tlačítkem a zvolit opět *Run As -> Java Application*. Textové výstupy programu jsou vypsány na konzoli. Pokud budeme program spouštět opakovaně, můžeme použít tlačítko v hlavní nástrojové liště nebo klávesu F11.

V případě, že během překladu nebo při spuštění byly zjištěny chyby, je nutné je opravit ve zdrojovém souboru a spustit překlad znovu. Nejčastější zdrojem chyb je, kromě běžných překlepů i skutečnost že Java je kontext senzitivní, takže je nutné dodržet velká i malá písmena, tak jak jsou uvedena v tomto příkladu a v dokumentaci Javy, včetně názvů souborů, které musí odpovídat přesně jménu třídy na řádku class ve zdrojovém textu.

Obrázek 18: Spouštění programu



3 ZÁKLADNÍ PRVKY JAZYKA JAVA

Jestliže jsme dosud nikdy nepoužívali objektově orientovaný jazyk, budeme se muset prvně naučit základní koncepty, než začneme psát první kód. Tato kapitola nás uvede do objektů, tříd, dědičnosti, rozhraní a balíčků. Každá část kapitoly ukazuje na to, jak jsou jednotlivé pojmy propojeny s reálným světem a zároveň nás uvede do základů programovacího jazyka Java.

3.1 ZÁKLADY SYNTAXE

Pro názornost si základy syntaxe jazyka Java ukážeme na nasledujícím příkladu.

PŘÍKLAD 2

```
import java.util.*;
public class Hello2
{
  static void Tisk(String s)
  { System.out.println(s);}

  public static void main(String args[])
  {
    String pozdrav;
    pozdrav="Ahoj dneska je "+ (new Date());
    Tisk(pozdrav);
  }
}
```

Jak je z příkladu vidět, pro syntaxi Javy platí určitá základní pravidla. Veškerá prováděná činnost i identifikátory programu je uložena v **definici tříd**, neexistují tedy žádné samostatné globální proměnné či funkce jako v jiných, plně neobjektových programovacích jazycích.

Zdrojový soubor, který má být spustitelný musí obsahovat veřejnou třídu uvozenou klíčovými slovy public class. Soubor musí být uložen pod jménem této třídy, s příponou .java (v příkladu Hello2.java).

Tato třída musí obsahovat metodu se záhlavím public static void main(String args[]), která se spouští při startu programu.

Kód programu je hierarchicky strukturován v blocích, uzavřených v složených závorkách {}. Nejvyšší úroveň je celá třída, která obsahuje kromě **deklarace svých atributů**, také samostatně uzavřené metody. Bloky na ještě nižší úrovně mohou uzavírat skupiny příkazů, které se mají zpracovat společně (například při splnění určité podmínky). Atributy třídy bývají často označovány jako **statické atributy**, protože jsou v programu uvozeny slovem static. Atributy třídy jsou pro všechny instance společné.

Příkazy jsou ve zdrojovém textu odděleny znakem ; (středník). Na řádku může být i více příkazů, ale zejména z důvodu ladění se doporučuje jen jeden.

Kromě příkazů zdrojový kód obsahuje **volání metod tříd** (ze standardních knihoven Javy (více kap. 11), třetích stran, nebo svých vlastních) ve formátech:

- metoda(*parametry*);
  Tak volá třída své vlastní metody (viz. metoda Tisk) .

- objektTridy.metoda(*parametry*);
  Standardní způsob volání metod. Objekt musí být předem vytvořen.
- trida.metoda(*parametry*);
  Tento způsob je povolen u tříd, které jsou označeny jako static (viz. System.out.println(s)).

Při použití třídy (nebo i jiného identifikátoru) z nějaké knihovny musíme překladači sdělit že tuto knihovnu používáme. Plně by tedy použití objektu Date mělo být new java.util.Date(), ale seznam použitých knihoven se obvykle uvádí společně do prvního řádku zdrojového kódu v příkazu import (vyjímku tvoří knihovna java.lang která se připojuje automaticky, proto jsme ji pro použití standardní metody pro výstup System.out.println nemuseli uvádět).

Identifikátory (názvy proměnných, metod, atributů, tříd, atd.) mohou obsahovat malá i velká písmena, číslice, znaky, podtržítko a $.Velká a malá písmena se považují za rozdílné znaky. První znak musí být písmeno nebo podtržítko.

Jazyk Java nic jako "hlavní program" nemá, místo toho však můžeme v libovolné třídě vytvořit metodu s názvem main(), kterou pak zavolá program Java po uvedení názvu třídy jako parametru. Jenže jak můžeme volat metodu, když jsme ještě nevytvořili instanci žádného objektu? K tomu slouží klíčové slovo static uvedené v hlavičce metody main()a označuje tzv. **třídní metodu**, tj. metodu, která se vztahuje k celé třídě a ne jen k jediné instanci. Podobně můžeme označit i proměnnou, která pak bude dostupná všem instancím jako třídní proměnná. Příkladem třídní proměnné je proměnná out ve třídě System, jejíž metodu println() jsme použili pro výpis na standardní výstup.

NEZAPOMEŇTE

Budu-li říkat, že *něco* **deklaruji**, znamená to, že někde veřejně vyhlašuji, jaké má dané *něco* vlastnosti, nebo se zavazuji, že splním to, co dané *něco* požaduje.

Řeknu-li naopak, že *něco* **definuji**, znamená to, že dané *něco* v programu zavádím, nechávam tomu přidělit paměť, přiřadit počáteční hodnotu apod.

3.2 Co je to Objekt?

Objekty jsou klíčem k pochopení technologie objektově orientovaného programování. Podíváme-li se nyní kolem sebe, jistě uvidíme spoustu příkladů objektů z reálného světa: váš pes, váš stůl, vaše televize, vaše kolo.

Reálné objekty sdílí dvě charakteristiky: všechny mají **stavy** a **akce**. Psi mají stavy (jméno, barva, rasa, hlad) a akce (štěkání, vrčení, vrtění ocasem). Jízdní kola mají taktéž stavy (momentální otáčky pedálu, zařazený stupeň, momentální rychlost) a akce (změna vybavení, přehození stupně, použití brzd). Identifikace stavů a akcí reálných objektů je dobrá cesta, jak začít myslet v rámci objektově orientovaného programování.

Nyní se chvíli dívejme kolem sebe a hledejme reálné objekty. Pro každý objekt, který uvidíme, si položme tyto dvě otázky: „Jaké možné stavy tento objekt může mít?“ a „Jaké možné akce může tento objekt vykonat?“ Vypišme si výsledky našeho pozorování. Poté, co to uděláme, si všimněme, že reálné objekty se velice liší ve své komplexnosti; vaše stolní lampa má pouze dva možné stavy (vypnuto a zapnuto) a dvě možné akce (zapnout a vypnout), ale vaše stolní rádio může mít mnohem více stavů (vypnuto, zapnuto, momentální hlasitost, naladěná stanice) a akcí (vypnout, zapnout, zesílit zvuk, zeslabit zvuk, najít stanici, změnit frekvenci). Možná najdeme

objekty, které obsahují jiné objekty. Tyto věci lze do softwaru přeložit pomocí objektově orientovaného programování.

Softwarové objekty sdílí stejný koncept jako objekty z reálného světa: taktéž obsahují stavy a příbuzné akce. Objekt ukládá své stavy ve **vlastnostech** (v některých programovacích jazycích se nazývají členské proměnné) a příslušné akce nabízí přes **metody** (v některých programovacích jazycích členské funkce). Metody operují nad interními stavy objektu a jsou předurčeny k prvotnímu způsobu komunikace mezi dvěma objekty. Skrývání interních stavů a jejich nabízení prostřednictvím metod se nazývá **zapouzdření dat,** jak již bylo zmíněno v předchozím textu. Vlastní objekty pak označujeme jako **instance** příslušné třídy.

NÁZORNÝ PŘÍKLAD

Ukažme si to na příkladu jízdního kola. Uchováváním stavů (momentální rychlost, zařazený stupeň a otáčky) a poskytováním metod pro jejich změnu zůstává objektům možnost kontrolovat, jak okolní svět zachází s hodnotami stavů. Například, pokud má objekt přehazovačku, která má pouze 6 stupňů, měla by metoda pro změnu zařazeného stupně odmítnout jakoukoli hodnotu, která je menší než 1 a větší než 6.

Zapouzdřením kódu do jednotlivých softwarových objektů získáme spoustu **výhod**, mezi které patří:

- **Modularita** – zdrojový kód objektu může být napsán a spravován nezávisle na zdrojovém kódu jiných objektů. Jakmile je jednou objekt vytvořen, může být jednoduše používán v celém programu.
- **Skrývání informací** – lepší udržovatelnost programu nám přináší skrývaní implementačních detailů dané třídy, což vynucuje nezávislost implementace ostatních tříd na implementaci aktuálni třídy.
- **Opětovné používání kódu** – jestliže již jednou třída/metoda existuje, můžeme tuto třídu/metodu snadno používat ve svém programu. Toto umožňuje vývojářům naprogramovat/otestovat/vyladit specifické třídy/metody, které potom můžeme bezpečně ve svém programu využít.
- **Rozšiřitelnost a snadné ladění** – jestliže začne nějaká třída/metoda vykazovat chyby, je možno tuto metodu jednoduš nahradit jinou. Toto je analogie s řešením některých praktických problémů. Například, jestliže se pokazí brzdy, vyměníme pouze je, ne celé auto.

3.3 CO JE TO TŘÍDA?

V reálném světě často najdeme příklady objektů, které jsou stejného druhu. Mohou existovat tisíce jízdních kol, které jsou jako jeden model vyrobeny stejně. Každé z jízdních kol bylo vyrobeno podle stejného návrhu a mají stejné komponenty. V objektově orientované terminologii můžeme říct, že právě jedno jízdní kolo je instancí jedné třídy objektů, známé jako jízdní kola. **Třída** je návrh, podle kterého jsou jednotlivé objekty vytvářeny. Následující třída Bicycle je jedna z možných implementací třídy jízdního kola:

PŘÍKLAD 3

```
class Bicycle {

    int cadence = 0;
```

```
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
        cadence = newValue;
    }

    void changeGear(int newValue) {
        gear = newValue;
    }

    void speedUp(int increment) {
        speed = speed + increment;
    }

    void applyBrakes(int decrement) {
        speed = speed - decrement;
    }

    void printStates() {
        System.out.println("cadence:"+cadence+"speed:"+speed+" gear:"+gear);
    }
}
```

Syntaxe programovacího jazyka Java nám nyní může připadat nová, ale návrh této třídy je založen na návrhu jízdního kola z předchozího příkladu. Vlastnosti cadence, speed a gear reprezentují možné stavy objektu a metody (changeCadence, changeGear, speedUp, a jiné) definují, jak bude objekt komunikovat s okolním světem.

Můžeme si všimnout, že třída Bicycle neobsahuje metodu main. To je z toho důvodu, protože se nejedná o úplnou aplikaci. Jjedná se pouze o návrh jízdního kola, který může být použit v aplikaci. Zodpovědnost za vytváření a používání nových objektů třídy Bicycle leží na nějaké jiné třídě naší aplikace.

Zde je třída BicycleDemo, která vytváří dva oddělené objekty třídy Bicycle a volá jejich metody:

PŘÍKLAD 4

```
class BicycleDemo {
    public static void main(String[] args) {

        // Create two different Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
```

```
    }
}
```

Výstup tohoto programu vypíše koncový stupeň, rychlost a vybavení pro dvě jízdní kola:
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3

Třída může mít obecně libovolný počet instancí. Existují však i třídy, které dovolí vytvoření pouze omezeného počtu instancí, někdy dokonce povolí jen jedinou instanci.

3.4 CO JE TO DĚDIČNOST?

Různé druhy reálných objektů mají spoustu společných vlastností s jinými druhy objektů. Horská, silniční a tandemová kola například sdílí vlastnosti jízdního kola (rychlost, přehazovačka, atd.). Každý druh kola však obsahuje další vlastnosti, které ho odlišují od ostatních: tandemová kola mají dvoje sedátka a řídítka; silniční kola mají odstranitelná sedátka; některá horská kola mají přídavný řetěz.

Objektově orientované programování umožňuje třídám **dědit** obecně používané stavy a akce od jiných tříd. V našem příkladě se třída Bicycle stane předkem (superclass) tříd MountainBike, RoadBike a TandemBike (viz další příklad). V programovacím jazyce Java je povoleno, aby každá třída měla pouze jednoho přímého předka, ale může mít neomezená množství potomků (subclasses). Syntaxe vytvoření potomka jiné třídy je snadné. Na začátku definice naší třídy uvedeme klíčové slovo extends následované jménem třídy, od které chceme potomka odvodit:

PŘÍKLAD 5

```
class MountainBike extends Bicycle {

    // zde budou přidané vlastnosti a metody

}
```

Tato konstrukce zajišťuje to, že třída MountainBike bude mít stejné vlastnosti a metody, jako má třída Bicycle. A zároveň můžeme přidat do této konstrukce kód, který ji učiní jedinečnou. Kód potomka se stává lépe čitelným. Nicméně, musíme se pečlivě starat o dokumentaci každé metody a vlastnosti v předkovi, neboť tyto vlastnosti nejsou součástí zdrojového kódu potomka.

3.5 CO JE TO ROZHRANÍ?

Jak jsme se již naučili, metody zajišřují komunikaci objektu s okolními objekty. Metody tvoří **rozhraní** s okolním světem. Analogie z reálného světa: například tlačítka na dálkovém ovladači televize tvoří rozhraní mezi námi a elektrickými obvody uvnitř televizoru. Pomocí tlačítka Power můžeme televizi vypnout a zapnout. Tradičně je rozhraní (interface) objektu tvořeno skupinou metod s prázdnými těly. V případě příkladu jízdního kola by rozhraní vypadalo takto:

PŘÍKLAD 6

```
class ACMEBicycle implements Bicycle {

  // zbytek třídy bude stejný jako výše

}
```

Tím, že ve třídě přidáme konstrukci implements zajistíme, že tato třída bude implementovat všechny metody, které toto rozhraní nabízí. Rozhraní definují vztah mezi třídou a ostatními třídami. Jestliže naše třída říká, že implementuje nějaké rozhraní, všechny metody, definované v rozhraní, se musí objevit ve zdrojovém kódu třídy předtím, než bude třída úspěšně zkompilována. Poznámka: Abysme jsme mohli zkompilovat třídu ACMEBycicle, budeme muset přidat klíčové slovo public na začátek implementace všech metod, definovaných rozhraním. Důvody se naučíme v příštích kapitolách.

3.6 CO JE TO BALÍČEK?

Balíček (package) je názvový prostor pro skupinu příbuzných tříd a rozhraní. Koncepčně je systém balíčků shodný se systémem složek na našem pevném disku. Můžeme uchovávat HTML stránky v jedné složce, obrázky ve druhé a skripty nebo aplikace v jiné. Protože se software napsaný v Javě může skládat ze stovek nebo dokonce tisíců jednotlivých tříd, je možnost udržet v nich přehled pomocí organizování příbuzných tříd a rozhraní do balíčků.

Java platforma poskytuje enormní knihovnu tříd (skupinu balíčků), určenou pro použití v našich aplikacích. Tato knihovna je známa jako API. Tyto balíčky reprezentují běžné úkony v programování. Například, objekt String reprezentují stavy a akce pro řetězce znaků; objekt File umožňuje vývojáři vytvářet, mazat, prohlížet nebo upravovat soubor na pevném disku; objekt Socket umožňuje vytvářet a používat síťové sokety; různé objekty UI, jako jsou tlačítka nebo seznamy, umožňují snadno vytvářet grafické uživatelské rozhraní. Existují doslova tisíce tříd, ze kterých si můžeme vybrat. To nám umožňuje zaměřit se přímo na vytváření aplikací.

Dokument *Java Platform API Specifikace* obsahuje seznam všech balíčků, tříd, rozhraní, vlastností a metod nabízených produktem Java Platform 6, Standard Edition.

3.7 IMPLEMENTACE TŘÍDY

Zde je příklad možné implementace třídy JizdniKolo pro získání přehledu o **deklaraci třídy**.

PŘÍKLAD 7

```
public class JizdniKolo {

   // JizdniKolo má tři atributy
   public int kadence;
   public int prevod;
   public int rychlost;

   // JizdniKolo má jeden konstruktor
   public JizdniKolo(int startovniKadence, int startovniRychlost, int startovniPrevod) {
      prevod = startovniPrevod;
```

```
        kadence = startovniKadence;
        rychlost = startovniRychlost;
    }

    // JizdniKolo má čtyři metody
    public void setKadence(int novaHodnota) {
        kadence = novaHodnota;
    }

    public void setPrevod(int novaHodnota) {
        prevod = novaHodnota;
    }

    public void brzdit(int kolik) {
        rychlost -= kolik;
    }

    public void zrychlit(int kolik) {
        rychlost += kolik;
    }

}
```

Deklarace třídy HorskeKolo jako potomka třídy JizdniKolo může vypadat nějak takto:

PŘÍKLAD 8

```
public class HorskeKolo extends JizdniKolo {

    // HorskeKolo přidává jednu atribut
    public int vyskaSedadla;

    // HorskeKolo má jeden constructor
    public HorskeKolo(int startovniVyska, int startovniKadence, int startovniRychlost, int
startovniPrevod) {
        super(startovniKadence, startovniRychlost, startovniPrevod);
        vyskaSedadla = startovniVyska;
    }

    // HorskeKolo přidává jednu metodu
    public void setVyska(int vyska) {
        vyskaSedadla = vyska;
    }

}
```

HorskeKolo zdědí všechny atributy a metody třídy JizdniKolo a přidává atribut vyskaSedadla a metodu pro nastavení (horská kola mají sedadlo, které se může pohybovat nahoru a dolů, podle současného terénu).

Třída JizdniKolo používá následující řádky kódu pro definici svých atributů:

```
public int kadence;
public int prevod;
public int rychlost;
```

Deklarace atributů se skládá ze tří částí, v tomto pořadí:

1. Žádný nebo více modifikátorů, jako je public nebo private.
2. Typ atributu.
3. Název atributu.

Atributy třídy JizdniKolo se nazývají kadence, prevod a rychlost a všechny jsou datového typu celé číslo int. Klíčové slovo public identifikuje tyto členy jako veřejné, čili přístupné pro kterýkoli objekt používající tuto třídu.

První (nejvíce vlevo) **modifikátor** umožňuje určit, které ostatní třídy mají přístup k danému atributu.

- Modifikátor public (veřejný) – atribut je přístupný ze všech tříd.
- Modifikátor private (soukromý) – atribut je přístuný pouze z aktuální třídy.

V případě zapouzdření se atributy nastavují jako soukromé. To znamená, že mohou být přímo přístupné pouze ze třídy JizdniKolo. Nicméně stále potřebujeme přístup k atributům. To může být uděláno nepřímo přidáním **veřejných metod**, které nám atributy zpřístupní:

PŘÍKLAD 9

```
public class JizdniKolo {

    private int kadence;
    private int prevod;
    private int rychlost;

    public JizdniKolo(int startovniKadence, int startovniRychlost, int startovniPrevod)  {
            prevod = startovniPrevod;
            kadence = startovniKadence;
            rychlost = startovniRychlost;
    }

    public int getKadence() {
            return kadence;
    }

    public void setKadence(int novaHodnota) {
            kadence = novaHodnota;
    }

    public int getPrevod() {
            return prevod;
    }

    public void setPrevod(int novaHodnota) {
            prevod = novaHodnota;
    }

    public int getrychlost() {
            return rychlost;
    }

    public void brzdit(int snizeni) {
            rychlost -= snizeni;
    }

    public void zrychlit(int zvyseni) {
            rychlost += zvyseni;
    }
```

```
}
```

3.8 VYTVOŘENÍ OBJEKTU

Typický program v jazyce Java vytváří hodně objektů, které mezi sebou, jak již víme, komunikují pomocí metod. Přes tyto interakce může program řešit různé úkoly. Jakmile objekt dokončí práci, pro kterou byl vytvořen, jsou jeho zdroje automaticky recyklovány pro použití v jiných objektech. Zde je malý ukázkový program, který vytvoří a použije tři objekty:

PŘÍKLAD 10

```
public class DemoVyytvoreniObjektu {

   public static void main(String[] args) {

      //Vytvoří objekt typu bod (Point)
      Point bodJedna = new Point(23, 94);
      //Vytvoří objekty typu obdelnik (MyRectangle)
      MyRectangle ctverecJedna = new MyRectangle(bodJedna, 100, 200);
      MyRectangle obdelnikDva = new MyRectangle(50, 100);

      //zobrazí informace o prvnim obdelniku
      System.out.println("Sirka prvniho obdelniku: " +
            ctverecJedna.sirka);
      System.out.println("Vyska prvniho obdelniku: " +
            ctverecJedna.vyska);
      System.out.println("Obsah prvniho obdelniku: " +
            ctverecJedna.getArea());

      //nastavi pozici druheho ctverce
      obdelnikDva.stred = bodJedna;

      //zobrazi pozici druheho ctverce
      System.out.println("X souradnice druheho obdelniku: "

            + obdelnikDva.stred.x);
      System.out.println("Y souradnice druheho obdelniku: "
            + obdelnikDva.stred.y);

      //presune druhy obdelnik a zobrazi jeho novou pozici
      obdelnikDva.move(40, 72);
      System.out.println("X souradnice druheho obdelniku: "
            + obdelnikDva.stred.x);
      System.out.println("Y souradnice druheho obdelniku: "
            + obdelnikDva.stred.y);
   }
}
```

Tento program vytváří různé objekty, manipuluje s nimi a zobrazuje je. Zde je jeho výstup:

```
Sirka prvniho obdelniku: 100
Vyska prvniho obdelniku: 200
Obsah prvniho obdelniku: 20000
X souradnice druheho obdelniku: 23
```

Y souradnice druheho obdelniku: 94
X souradnice druheho obdelniku: 40
Y souradnice druheho obdelniku: 72

3.8.1 VÍCE O VYTVOŘENÍ OBJEKTŮ

Jak již víme, třídy vytvářejí vzor pro objekty; dle třídy poté můžeme vytvořit objekt. Každý z následujících řádků vytváří jeden objekt:

PŘÍKLAD 11

```
    Point bodJedna = new Point(23, 94);
//Vytvoří objekty typu obdelnik (MyRectangle)
MyRectangle ctverecJedna = new MyRectangle(bodJedna, 100, 200);
MyRectangle obdelnikDva = new MyRectangle(50, 100);
```

První řádek vytváří objekt třídy Point (bod) a další dva vytváří objekty tříd Rectangle (obdélník). Každý z těchto příkazů má tři části (o kterých se budeme dále bavit):

- **Deklarace –** kódy, které jsou **tučně**, jsou deklarace proměnných, jež přiřazují jména proměnných odpovídajícím typům.
- **Vytvoření instance** – klíčové slovo new je operátor Javy, který vytvoří nový objekt.
- **Inicializace** – operátor new je následován voláním konstruktoru (více o konstruktorech v kapitole 7), který inicializuje třídu.

Když se snažíme deklarovat proměnnou, tak píšeme:

typ název;

To upozorní kompilátor, že budeme používat název pro odkazovaní na data, jejichž typ je typ. Můžeme také deklarovat proměnnou referenčního typu na vlastním řádku. Například:

Point prvniBod;

Pokud jsme deklarovali prvniBod jako výše, jeho hodnota bude neurčená, dokud nebude objekt vytvořen a této proměnné přiřazen. Jednoduchá deklarace proměnné referenčího typu nevytvoří objekt. Pro vytvoření musíme použít operátor new, jak je popsáno v další podkapitole. Tento objekt musíme do originOne přiřadit dříve, než proměnnou ve své aplikaci použijeme. Jinak získáme chybu kompilátoru.

3.8.2 VYTVOŘENÍ INSTANCE TŘÍDY

Operátor new specifikuje třídu, alokuje potřebnou paměť a vrátí odkaz na tuto paměť. Operátor new také zavolá konstruktor třídy. Fráze „instancovat třídu“ znamená to samé jako „vytvořit objekt“. Když vytváříme objekt, vytváříme „instanci“ třídy.

Operátor new požaduje jeden argument psaný za jeho názvem – volání konstruktoru. Operátor new vrací odkaz na nově alokovanou paměť. Tento odkaz je obvykle přiřazen proměnné odpovídajícího typu:

Point prvniBod = new Point(23, 94);

3.8.3 INICIALIZACE OBJEKTU

Zde je kód třídy Point(bod):

PŘÍKLAD 12

```
public class Point {
   public int x = 0;
   public int y = 0;
   // konstruktor
   public Point(int a, int b) {
         x = a;
         y = b;
   }
}
```

Tato třída obsahuje jeden konstruktor. Konstruktor poznáme podle toho, že má stejné jméno jako třída a že nemá návratový typ. Konstruktor ve třídě Point přebírá dva celočíselné argumenty, jak je deklarováno kódem (int a, int b). Následující příkaz poskytuje 23 a 94 jako hodnoty pro tyto argumenty:

```
Point bodPrvni = new Point(23, 94);
```

Zde je kód třídy MyRectangle (obdélník), která obsahuje 4 konstruktory:

PŘÍKLAD 13

```
public class MyRectangle {
   public int sirka = 0;
   public int vyska = 0;
   public Point stred;

   // čtyři konstruktory
   public MyRectangle() {
         stred = new Point(0, 0);
   }
   public MyRectangle(Point p) {
         stred = p;
   }
   public MyRectangle(int s, int v) {
         stred = new Point(0, 0);
         sirka = s;
         vyska = v;
   }
   public MyRectangle(Point p, int s, int v) {
         stred = p;
         sirka = s;
         vyska = v;
   }

   // metoda pro pohyb s obdelnikem
   public void move(int x, int y) {
         stred.x = x;
         stred.y = y;
   }

   // metoda pro vypocet obsahu obdelniku
   public int getArea() {
```

```
        return sirka * vyska;
    }
}
```

Každý konstruktor nám umožňuje nastavit střed, výšku a šířku obdélníku, užívajíce jak primitivní typy, tak referenční. Jestliže má třída více konstruktorů, musí mít odlišné **signatury**. Java kompilátor odlišuje konstruktory podle počtu typu argumentů.

Když se Java kompilátor setká s následujícícm kódem, ví, že má zavolat konstruktor ve třídě MyRectangle, který požaduje argument Point následovaný dvěma celočíselnými argumenty:

```
MyRectangle ctverecJedna = new MyRectangle(bodJedna, 100, 200);
```

Tento kód zavolá jeden z konstruktorů MyRectangle, který inicializuje stred na prvniBod. Dále konstruktor nastaví šířku na 100 a výšku na 200. Nyní jsou dva odkazy na stejný objekt Point.

Následující řádek kódu zavolá konstruktor třídy MyRectangle, který vyžaduje dva celočíselné argumenty, které poskytují inicializační hodnoty pro šířku a výšku. Jestliže si prohlédneme kód uvnitř konstruktoru, uvidíme, že vytváří nový objekt Point, jehož hodnoty x a y jsou inicializovány na 0:

```
MyRectangle obdelnikDva = new MyRectangle(50, 100);
```

Konstruktor MyRectangle, používaný v následujícím příkazu nepřebírá žádné argumenty, takže je nazýván bezargumentový (výchozí) konstruktor:

```
MyRectangle rect = new MyRectangle();
```

3.8.4 UŽÍVÁNÍ OBJEKTŮ

Atributy objektu jsou přístupné pomocí názvu, který musí být jednoznačný. Pokud jsme přímo v dané třídě, můžeme pro atributy použít jednoduše její jméno. Například můžeme přidat příkaz uvnitř kódu třídy MyRectangle, který vytiskne výšku a šířku:

```
System.out.println("Vyska a sirka jsou: " + vyska + ", " + sirka);
```

Kód, který je mimo třídu objektu, musí použít referenční proměnnou nebo výraz následovaný operátorem tečka (.) a až poté jednoduchým jménem atributu jako např. zde:

```
promennaSObjektem.nazevVlastnosti
```

Kód, který jsme uvedli v příkladu 10, byl mimo třídu MyRectangle. Proto při odkazování na atributy stred, sirka a vyska uvnitř objektu třídy MyRectangle nazvaného ctverecJedna musel program použít ctverecJedna.stred, ctverecDva.sirka a ctverecDva.vyska. Následující kousek programu využívá tyto jména k vytištění sirka a vyska objektu ctverecJedna:

```
System.out.println("Sirka prvniho obdelniku: " +
                   ctverecJedna.sirka);
System.out.println("Vyska prvniho obdelniku: " +
                   ctverecJedna.vyska);
```

Později používá program stejný kód pro zobrazení informací o obdelnikDva. Objekty stejného typu mají svou vlastní sbírku instančních atributů. Takže každý objekt třídy MyRectangle má atributy nazvané stred, sirka a vyska. Když k této instanci přistupujeme přes odkaz (např. proměnnou), ukazuje na příslušnou sadu

atributů. Tyto dva objekty ctverecJedna a obdelnikDva v programu z úvodní části mají odlišné atributy stred, sirka a vyska.

Pro přístup k atributům můžeme využít buďto referenční proměnnou, jako v minulých příkladech, nebo můžeme využít výraz vracející odkaz na objekt. Zavolání operátoru new vrací odkaz na objekt. Takže můžeme použít hodnotu vrácenou z operátoru k přístupu k atributům:

```
int vyska = new MyRectangle().vyska;
```

Tento příkaz vytvoří objekt MyRectangle a přímo přistupuje k atributu vyska. V podstatě tento příkaz vráti výchozí výšku objektu. Všimněme si, že jakmile se tento příkaz vykoná, program nebude mít dále odkaz na vytvořený objekt MyRectangle, protože program tento odkaz nikde neukládá.

3.8.5 VOLÁNÍ METOD OBJEKTU

Odkaz na objekt můžeme taktéž využít k **volání metod**. Prostě přidáme jednoduché jméno metody za odkaz na objekt, kterému předchází operátor tečka (.). Také poskytneme v kulatých závorkách potřebné argumenty, jsou-li nějaké. Jestliže metoda nevyžaduje žádné argumenty, použijeme prázdné kulaté závorky.

```
odkazNaObjekt.nazevMetody(argumenty);
odkazNaObjekt.nazevMetody();
```

Třída MyRectangle má dvě metody: getArea() pro výpočet plochy obdélníku a move()pro přesun středu obdélníku. Zde je kód z úvodní části, který volá metodu getArea():

```
System.out.println("Plocha prvniho obdelniku: " + ctverecJedna.getArea());
    ...
obdelnikDva.move(40, 72);
```

První příkaz zavolá metodu objektu ctverecJedna nazvanou getArea() a zobrazí výsledky. Druhý příkaz přesune obdelnikDva, protože metoda move() nastaví nové hodnoty atributům stred.x a stred.y.

Jako u atributů, odkazNaObjekt musí být platný odkaz na objekt. Můžeme použít název proměnné nebo jakýkoli jiný výraz vracející odkaz na objekt. Operátor new vrací odkaz na objekt, takže můžeme použít například následující kód:

```
new Rectangle(100, 50).getArea()
```

Výraz new Rectangle(100, 50) vrací odkaz, ukazující na objekt Rectangle. Některé metody, jako například getArea()vracejí hodnotu. Metody vracející hodnotu můžete použít ve výrazu.

Můžeme hodnotu přiřadit proměnné, můžeme podle hodnoty dělat rozhodnutí nebo kontrolovat smyčku. Následující kód přiřadí hodntu z metody getArea() do proměnné povrchObdelniku:

```
int povrchObdelniku = new Rectangle(100, 50).getArea();
```

Některé objektově orientované jazyky (např. C++) po programátorech požadují, aby objekty, které již nejsou využívány, byly odstraněny. Platforma Java nám umožňuje vytvořit tolik objektů, kolik chceme (počet může být samozřejmě omezen operačním systémem), a my se nemusíme starat o jejich odstranění. Běhové prostředí Javy objekt odstraní, jakmile zjistí, že již nebude dále potřeba. Tento proces je nazýván **garbage collection** (sběr odpadů) a stará se o něj **garbage collector** (doslova přeloženo popelář).

Objekt bude odstraněn, jakmile nebude existovat žádný odkaz na tento objekt. Odkazy držené v proměnných jsou obvykle ztraceny, jakmile jsou proměnné mimo svůj obor platnosti. Nebo můžeme explicitně odstranit odkaz, když proměnné přiřadíme speciální hodnotu null. Pamatujme, že program může mít více odkazů na stejný objekt; všechny odkazy musí být odstraněny před tím, než bude odstraněn objekt.

Běhové prostředí Javy má garbage collector, který periodicky recykluje objekty, na které neexistuje žádný odkaz. Garbage collector dělá tuto práci automaticky, jakmile uzná, že je vhodný čas.

4 PŘÍKAZY JAZYKA JAVA

4.1 PROMĚNNÉ

V předchozí kapitole jsme se již naučili, že objekt ukládá své stavy ve vlastnostech. Nicméně, programovací jazyk Java používá taktéž termín **proměnné**. Tento díl pojednává o jejich závislostech, včetně pravidel a konvencí pro pojmenovávání proměnných, o základních datových typech (primitivní typy, řetězce znaků a pole), výchozích hodnotách a literálech.

V programovacím jazyce Java máme následující druhy proměnných:

- **Proměnné instance** (nestatické vlastnosti) - technickou mluvou, objekty ukládají své individuální stavy do „nestatických vlastností“, což jsou vlastnosti definované bez klíčového slova static. Nestatické vlastnosti jsou taktéž známy jako proměnné instance, protože jejich hodnoty jsou jedinečné pro každou instanci třídy (jinými slovy, pro každý objekt); vlastnost currentSpeed z Příkladu 5 je nezávislá na vlastnosti currentSpeed jiného.
- **Proměnné třídy** (statické vlastnosti) - Proměnné třídy jsou vlastnosti, u nichž byl uveden modifikátor static; tento modifikátor říká kompilátoru, že tato vlastnost má existovat pouze jednou, nezávisle na tom, kolik objektů této třídy bylo vytvořeno. Vlastnost definující počet převodů na kole může být definována s klíčovým slovem static, protože počet převodů mají všechna kola stejný. Kód static int numGears = 6; tuto statickou vlastnost vytvoří. Případně může být předáno klíčové slovo final, aby bylo zaručeno, že se v průběhu programu počet převodů nezmění.
- **Lokální proměnné** - stejně jako objekt ukládá své stavy do vlastností, metoda ukládá své dočasné stavy do lokálních proměnných. Syntaxe deklarující lokální proměnnou je naprosto stejná jako deklarace vlastnosti (např. int count = 0;). Neexistuje žádné klíčové slovo říkající, že proměnná je lokální; kompilátor to pozná podle umístění deklarace. Lokální proměnná se deklaruje uvnitř složených závorek, ohraničujících metodu. Z toho navíc vyplývá, že lokální proměnné může využít pouze metoda, která je definuje. Žádná jiná část třídy k nim nemá přístup.
- **Parametry** - jsme již ve třídě Bicycle a v metodě main „Hello World!“ aplikace. Podíváme-li se znovu na deklaraci metody main, tato metoda je deklarována jako public static void main(String[] args). Zde je proměnná args parametrem této metody. Je důležité si uvědomit, že parametry jsou kvalifikovány jako „proměnné“, ne jako „vlastnosti“. Toto se týká také jiných částí kódů přebírajících parametry (jako jsou výjimky a konstruktory, o kterých si budeme povídat v dalších kapitolách).

Je důležité si tyto pojmy zapamatovat, budeme je v tomto kontextu dále používat. Dále budeme používat termín **členská data (členové)**. Vlastnosti, metody a vložené typy jsou kolektivně nazývané členská data třídy/objektu.

4.1.1 POJMENOVÁVÁNÍ

Každý programovací jazyk má svá pravidla a konvence pro pojmenovávání a ani u Javy tomu není jinak. Tato pravidla a konvence by se dala shrnout asi takto:

- U názvů proměnných záleží na velikosti písmen. Název proměnné se může skládat z libovolného platného znaku – délkově neomezená sekvence Unicode znaků a čísel – které musí začínat písmenem, znakem dolaru „$“ nebo znakem podtržítka „_“. Konvencí však je, že jméno proměnné vždy začíná písmenem, nikoli dolarem či podtržítkem. Můžeme najít situace, kdy automaticky generované názvy obsahují znak dolaru, ale jména našich proměnných by ho používat neměla. Stejná konvence platí i pro používání znaku podtržítko. Přestože je technicky možné použít na začátek proměnné znak „_“, není tato praktika doporučována. Prázdné místo není povoleno.
- Následující znaky mohou být písmena, číslice, znaky dolaru nebo podtržítka. Konvence je stejná jako u prvních písmen. Když vybíráme jméno pro naši proměnnou, používejme plná jména než kryptografické zkratky. Snažme se, co to jde, aby náš kód byl snadno čitelný. Často to vytváří kód, který sám říká, co dělá; např. jména vlastností cadence, speed a gear, jsou mnohem více srozumitelné, než jejich zkratky, jako jsou c, s a g. Mějme taktéž na paměti, že náš název **nesmí být nějaké již existující klíčové nebo slovo rezervované programovacím jazykem**.
- Jestliže se název naší proměnné skládá pouze z jednoho slova, pišme toto jméno celé malými písmeny. Jestliže obsahuje více než jedno slovo, zvětšeme každé první písmeno následujícího slova. Názvy gearRatio a currentGear jsou ukázkovými příklady této konvence. Jestliže naše vlastnost obsahuje konstantní hodnotu, jako je static final int NUM_GEARS = 6, konvence je mírně jiná, zvětšeme každé písmenko a slova oddělujte podtržítkem. Podle konvence se podtržítko nepoužívá nikde jinde.

4.1.2 PRIMITIVNÍ DATOVÉ TYPY

Programovací jazyk Java je silně typový, což znamená, že všechny proměnné musí být prvně **deklarovány**, než budou moci být použity. Toto zahrnuje zadání typu a jména proměnné, jak můžeme vidět zde:

```
int gear = 1;
```

Tímto říkáme, že náš program má vlastnost „gear“ uchovávající číselná data a její výchozí hodnota je „1“. Typ proměnné definuje, jaké hodnoty může proměnná uchovávat a jaké operace mohou být nad proměnnou vykonávány.

Kromě typu int podporuje programovací jazyk Java dalších 7 primitivních datových typů. Primitivní typ je předdefinován jazykem a je pojmenován rezervovaným klíčovým slovem. Primitivní hodnoty nesdílí své stavy s ostatními hodnotami.

Mezi osm primitivních datových typů, které jsou poskytovány programovacím jazykem Java, patří:

- byte: Datový typ byte obsahuje 8bitové dvakrát doplněné celé číslo se znaménkem. Jeho minimální hodnota je −128 a maximální 127. Datový

typ byte může být použit pro ušetření paměti ve velkých polích, kde se může ve speciálních případech hodit každý bit.

- short: Datový typ short je 16bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −32 768 a maximální 32 767. Stejně jako u typu byte jsou zde stejné případy použití: typ short můžete použít k ušetření paměti při vytváření velkých polí u náročných aplikací.
- int: Datový typ int je 32bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −2 147 483 648 a maximální 2 147 483 647. Pro celočíselné hodnoty je tento typ tou základní volbou, pokud se nedostanete do výše popsaných situací. Tento datový typ bude dostatečně velký pro většinu čísel ve vaší aplikaci, pokud však budete potřebovat širší rozsah hodnot, použijte místo něj typ long.
- long: Datový typ long je 64bitové dvakrát doplněné celé číslo. Jeho minimální hodnota je −9 223 372 036 854 775 808 a maximální 9 223 372 036 854 775 807. Používejte tento datový typ, pokud vám nebude stačit rozsah hodnot poskytovaný datovým typem int.
- float: Datový typ float je 32bitové číslo s pohyblivou desetinou čárkou. Stejně jako u typu byte a short používáme typ float (místo double), pokud potřebujeme ušetřit paměť ve velkých polích. Tento datový typ bychom nikdy neměli používat pro přesné hodnoty, jako jsou finanční částky. Pro tento případ budeme muset použít třídu java.math.BigDecimal.
- double: Datový typ double je 64bitové číslo s pohyblivou desetinou čárkou. Pro desetinná čísla je tento datový typ tou základní volbou. Stejně jako typ float se double nehodí pro přesné hodnoty, jako jsou finanční částky.
- boolean: Datový typ boolean má povoleny pouze 2 hodnoty: true a false. Používáme tento datový typ, abychom specifikovali, zda je podmínka pravdivá (true) nebo nepravdivá (false). Tento datový typ obsahuje 1 bit informací, ale jeho velikost není přesně definována.
- char: Datový typ char je jednoduchý 16bitový Unicode znak. Jeho minimálni hodnotu je u0000 (nebo 0) a maximální uffff (nebo 65 535).

Kromě 8 primitivních datových typů nabízí programovací jazyk Java speciální podporu pro řetězce znaků, a to přes třídu java.lang.String. Zadáním vašeho řetězce do dvojitých uvozovek se automaticky vytvoří objekt třídy String; například, String mujString = "toto je řetězec";. Objekty String jsou neměnné, což znamená, že jakmile jsou jednou vytvořeny, jejich hodnoty se nemohou změnit.

4.1.3 VÝCHOZÍ HODNOTY

Není vždy nezbytně nutné přiřadit hodnotu vlastnosti hned při deklaraci. Vlastnosti, které byly deklarovány a neinicializovány, dostanou kompilátorem výchozí hodnotu. Obecně řečeno, do vlastností bude uložena buďto nula nebo null (pojem null bude vysvětlen později v textu), záleží na datovém typu. Avšak nechat vše závislé na výchozích hodnotách není příliš dobrý styl programování, obecně se doporučuje výchozí hodnotu vlastnosti napsat hned při deklaraci.

Následující tabulka ukazuje výchozí hodnoty pro výše uvedené datové typy.

Tabulka 2: Výchozí hodnoty pro vlastnosti

Datový typ	Výchozí hodnota (pro vlastnosti)
byte	0
short	0
int	0
long	0
float	0.0f
double	0.0d
char	'u0000'
String	null
Jakýkoli objekt	null
boolen	false

Lokální proměnné jsou mírně odlišné; kompilátor nikdy nepřiřadí výchozí hodnotu neinicializované lokální proměnné. Jestliže nemůžeme inicializovat proměnnou při její deklaraci, musíme zajistit, že tuto hodnotu proměnné přiřadíme dříve, než se pokusíme proměnnou použít. Přístup k neinicializované lokální proměnné vyústí v chybu při kompilaci.

4.1.4 LITERÁLY

Čtenář si již možná všiml, že když jsme inicializovali hodnotu primitivního datového typu, nepoužili jsme operátor new. Primitivní typy jsou speciální datové typy zabudované do jazyky; nejedná se o objekty třídy. **Literál** je reprezentace fixní hodnoty ve zdrojovém kódu; literály jsou přímou reprezentací hodnot bez dalších výpočtů. Jak je ukázáno níže, literál je možno uložit do proměnné primitivního datového typu:

```
boolean result = true;
char capitalC = 'C';
byte b = 100;
short s = 10000;
int i = 100000;
```

Hodnoty celočíselných datových typů (byte, short, int a long) mohou být zapsány pomocí desítkové, osmičkové nebo šestnáctkové soustavy. Desítkovou soustavu používáte každý den; je založena na 10 číslicích (od 0 do 9). Osmičková soustava obsahuje 8 čísel (od 0 do 7) a konečně šestnáctková (hexadecimální) je založena na číslech od 0 do 9 a navíc písmenech A až F. Dá se říct, že při běžném programování budete používat pouze desítkovou soustavu. Nicméně, pokud budete potřebovat osmičkovou nebo šestnáctkovou soustavu, následující příklad ukazuje správnou syntaxi. Prefix 0 značí osmičkovou, zatímco 0x značí šestnáctkovou.

```
int decVal = 26;      // číslo 26 v desítkové
int octVal = 032;     // číslo 26 v osmičkové
int hexVal = 0x1a;    // číslo 26 v šestnáctkové
```

Typy s pohyblivou desetinnou čárkou (float a double) mohou být inicializovány výrazem s E nebo e (pro vědecký zápis), F nebo f (32-bit float literál) a D nebo d (64-bit double literál; tento je výchozí a podle konvence se vynechává).

```
double d1 = 123.4;
double d2 = 1.234e2; // vědecký zápis
float f1  = 123.4f;
```

Literály typů char a String mohou obsahovat jakékoli Unicode (UTF-16) znaky. Jestliže to vaše prostředí umožňuje, můžete znaky zapisovat přímo. Můžeme použít tzv. „Unicode escape“, jako je např. 'u0108' (velké C s circumflexem) nebo "Su00ED seu00F1or" (Sí señor ve španělštině). Vždy používejme 'jednoduché uvozovky' pro char literály a "dvojité uvozovky" pro String literály. Unicode escape sekvence mohou být použity kdekoli v programu (jako jsou např. názvy vlastností), ne pouze v char nebo String literálech.

Programovací jazyk Java taktéž podporuje několik speciálních escape sekvencí pro char a String literály: b (backspace), t (tabulátor), n (nový řádek), f (form feed), r (návrat vozíku), " (dvojité uvozovky), ' (jednoduché uvozovky) a \ (zpětné lomítko).

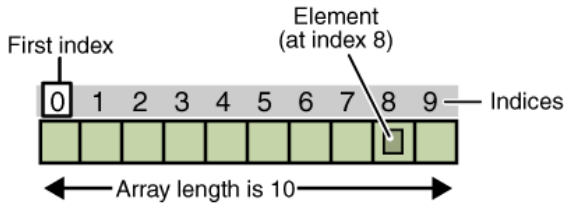
Taktéž existuje speciální literál null, který může být přiřazen proměnné libovolného referenčního typu. null může být přiřazen libovolné proměnné kromě těch, které jsou primitivního typu. Je toho málo, co můžete s hodnotou null dělat kromě testování toho, zda je nastavena. Proto je často null používán k indikaci, že je nějaký objekt nedostupný.

Konečně, je zde speciální druh literálu nazývaný **literál třídy** a zapisovaný názvem typu s přidáním .class; například String.class. Tento zápis reprezentuje objekt třídy Class, která obsahuje informace o typu.

4.1.5 POLE

Objekt pole je kontejner (pojem kontejner bude popsán v další podkapitole), který ukládá předem pevně daný počet položek stejného typu. Délka (počet prvků) pole je určena, když se pole vytváří. Po jeho vytvoření je jeho délka neměnitelná. Již jste viděli pole v akci, a to v metodě main() „Hello World!“ aplikace. Tato část textu popisuje pole do většího detailu.

Obrázek 19: Pole s deseti prvky



Každá položka v poli se nazývá **element** (**prvek**) a každý prvek je dostupný skrz jeho **číselný index**. Jak je ukázáno na ilustraci výše, začíná číslování na 0, devátý element je dostupný pod indexem 8.

Následující program ArrayDemo vytvoří pole celočíselných hodnot, nějaké do něj vloží a vytiskne je.

PŘÍKLAD 14

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] nejakePole; //deklaruje pole celých čísel

        nejakePole = new int[10];      // alokuje paměť pro 10 elementů

        nejakePole[0] = 100; // inicializuje první element
        nejakePole[1] = 200; // inicializuje druhý element
        nejakePole[2] = 300; // atd.
        nejakePole[3] = 400;
        nejakePole[4] = 500;
        nejakePole[5] = 600;
        nejakePole[6] = 700;
        nejakePole[7] = 800;
        nejakePole[8] = 900;
        nejakePole[9] = 1000;

        System.out.println("Element s indexem 0: " + nejakePole[0]);
        System.out.println("Element s indexem 1: " + nejakePole[1]);
        System.out.println("Element s indexem 2: " + nejakePole[2]);
        System.out.println("Element s indexem 3: " + nejakePole[3]);
        System.out.println("Element s indexem 4: " + nejakePole[4]);
        System.out.println("Element s indexem 5: " + nejakePole[5]);
        System.out.println("Element s indexem 6: " + nejakePole[6]);
        System.out.println("Element s indexem 7: " + nejakePole[7]);
        System.out.println("Element s indexem 8: " + nejakePole[8]);
        System.out.println("Element s indexem 9: " + nejakePole[9]);
    }
}
```

V reálné situaci bychom nejspíše použili jednu z dostupných smyček pro iteraci přes každý prvek pole než vypisování každého řádku zvlášť, jak je ukázáno výše. Nicméně, tento příklad jednoduše ilustruje syntaxi použití polí. Více se o různých smyčkách (for, while a do-while) naučíme v další části této kapitoly.

Výše uvedený program deklaruje proměnnou pro pole (nejakePole) následujícím řádkem kódu:

```
int[] nejakePole;   // deklaruje pole celých čísel
```

Stejně jako deklarace jakékoliv jiné proměnné, skládá se deklarace pole ze dvou částí: **typ pole** a **název pole**. Typ pole je napsán jako type[], kde type je datový typ uchovávaných hodnot; hranaté závorky jsou speciální zápis, který říká, že proměnná je pole. Velikost pole není součástí typu (to je důvod, proč jsou závorky prázdné). Název pole může být cokoliv, co splňuje pravidla a konvence pro pojmenovávání proměnných, o kterých jsme mluvili dříve. Stejně jako deklarace proměnných jiného typu, deklarace nevytvoří pole – jednoduše kompilátoru říká, že proměnná bude obsahovat pole hodnot určitého typu.

Stejně můžeme deklarovat pole jiných typů:

```
byte[] anArrayOfBytes;
short[] anArrayOfShorts;
long[] anArrayOfLongs;
float[] anArrayOfFloats;
```

```
double[] anArrayOfDoubles;
boolean[] anArrayOfBooleans;
char[] anArrayOfChars;
String[] anArrayOfStrings;
```

Taktéž můžete umístit hranaté závorky za název pole:

```
float anArrayOfFloats[]; // toto forma není doporučována
```

Nicméně, konvence zavrhuje tento typ zápisu; hranaté závorky definují typ proměnné, a proto by měly být uvedeny spolu s typem hodnot.

Jednou z cest k vytvoření pole je operátor new. Následující program ArrayDemo alokuje paměť pro 10 celočíselných hodnot a uloží toto pole do proměnné nejakePole.

```
nejakePole = new int[10]; // alokuje paměť pro 10 elementů
```

Kdyby tento příkaz chyběl, kompilátor by vydal tuto hlášku a kompilace by selhala:

```
ArrayDemo.java:4: Variable anArray may not have been initialized.
```

Následujích několik řádek toto pole inicializuje:

```
nejakePole[0] = 100; // inicializuje první element
nejakePole[1] = 200; // inicializuje druhý element
nejakePole[2] = 300; // atd.
```

Ke každém elementu v poli je přistupováno pomocí indexů:

```
System.out.println("Element s indexem 0: " + nejakePole[0]);
System.out.println("Element s indexem 1: " + nejakePole[1]);
System.out.println("Element s indexem 2: " + nejakePole[2]);
```

Kromě tohoto způsobu můžete vytvořit a inicializovat pole následujícím zkráceným zápisem:

```
int[] nejakePole = {100, 200, 300, 400, 500, 600, 700, 800, 900, 1000};
```

Zde je délka pole odvozena od počtu hodnot mezi { a }.

Můžete taktéž vytvářet pole polí (jsou taktéž známá jako **vícerozměrná pole**) použitím dvou nebo více dvojic hranatých závorek, jak např. String[][] jmena. Poté se musí ke každému elementu přistupovat s odpovídajícím počtem indexů.

V programovacím jazyce Java jsou vícerozměrná pole jednoduše pole obsahující další pole. Toto je rozdíl oproti polím v jazycích C nebo Fortran. Následkem této vlastnosti jazyka je, že vícerozměrná pole nemusí mít stejnou délku u všech rozměrů, jak ukazuje následující příklad:

PŘÍKLAD 15

```
class MultiDimArrayDemo {
   public static void main(String[] args) {
      String[][] jmena = {{"Mr. ", "Mrs. ", "Ms. "},
                          {"Smith", "Jones"}};
      System.out.println(jmena[0][0] + jmena[1][0]); //Mr. Smith
      System.out.println(jmena[0][2] + jmena[1][1]); //Ms. Jones
   }
}
```

Tento program vypíše následující řádky:

Mr. Smith
Ms. Jones

Můžeme také použít zabudovanou vlastnost length k zjištění velikosti pole. Kód System.out.println(nejakePole.length); vypíše délku pole na standardní výstup.

Třída System obsahuje metodu arraycopy, pomocí které můžete efektivně kopírovat data z jednoho pole do druhého:

```
public static void arraycopy(Object src,
                             int srcPos,
                             Object dest,
                             int destPos,
                             int length)
```

Dva parametry Object specifikují pole, ze kterého se bude kopírovat, a pole, do kterého se bude kopírovat. Tři int specifikují startovní pozici ve zdrojovém poli, startovní pozici v cílovém poli a počet elementů, které se zkopírují.

Následující program ArrayCopyDemo deklaruje pole elementů typu char hláskující slovo „decaffeinated“. Používá metodu arraycopy ke zkopírování části jednoho pole do druhého:

PŘÍKLAD 16

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] kopirovatZ = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
      char[] kopirovatDo = new char[7];

      System.arraycopy(kopirovatZ, 2, kopirovatDo, 0, 7);
      System.out.println(new String(kopirovatDo));
   }
}
```

Výpis programu:

Caffein

4.2 DYNAMICKÉ DATOVÉ STRUKTURY

Java poskytuje několik možností pro uložení většího množství dat (tj. objektů či primitivních datových typů) v paměti. S nejjednodušší z nich, s polem, jsme se právě seznámili. Pole je struktura velmi jednoduchá na používání, která má následující výhody:

- pole je ze všech datových struktur nejefektivnější z hlediska ukládání a vybírání dat, pokud však potřebuje provádět složitější operace (např. zajistit, aby prvek byl vložen pouze jednou), může být použití jiných struktur efektivnější,
- pole zajišťuje typovou kontrolu – pokud nadefinujete pole objektů určité třídy, můžete do něho vkládat pouze instance této třídy (a instance potomků této třídy),
- pole umožňuje vkládat přímo primitivní datové typy, u ostatních datových struktur se musí převést na objekty.

Pole má však také dvě velké nevýhody:

- je třeba předem znát počet prvků, které do něj budete ukládat,
- nepodporuje některé složitější způsoby práce s objekty (např. vkládání prvků doprostřed pole či vytváření asociativních polí).

Java poskytuje pro práci s poli pomocnou třídu Arrays, která nabízí např. třídění pole či prohledávání pole.

4.2.1 KONTEJNERY

Při ukládání objektů do dynamických datových struktur (často se pro ně používá pojem kontejner) nemusíme dopředu znát počet vkládaných objektů. Jednotlivé typy kontejnerů dále umožňují složitější operace s daty. Ve zbytku této kapitoly se budeme zabývat kontejnery implementovanými v Javě od verze 1.2. Starší verze Javy poskytují jiné dynamické struktury (Vector, Stack, HashTable) – tyto zde nebudeme popisovat. My se budeme věnovat dvěma významým třídám, ze který dále vycházejí ostatní třídy odpovídající dalším dynamickým datovým strukturám – **kolekce** (Collection) a **mapy** (Map).

Rozhraní Colletion

Rozhraní Collection (pojem rozhraní bude podrobně vysvětlen v Kapitole 9). Prozatím se spokojíme se zjednodušeným vysvětlením, že rozhraní je typ třídy, jejíž některé metody ještě nejsou naprogramovány, a pokud chce tuto třídu/rozhraní používat, musí všechny tyto předem stanovené a dosude nenaprogramované metody naprogramovat) poskytuje základní metody pro tvorbu a použití seznamů (List) a množin (Set). Seznamy ukládají instance do lineární struktury s opakováním prvků. Obvykle se používá ArrayList, pouze v případě velkého množství vkládání/vyjímání dovnitř/zevnitř seznamu je výhodnější LinkedList. Množiny ukládají pouze instance různých hodnot – pro zjištění odlišnosti se používá metoda equals (v praxi se do množiny vkládají instance pouze jedné třídy). Efektivnější je použití varianty HashSet, pouze v případě, že potřebujeme mít seznam hodnot trvale setříděn, používá se TreeSet.

Rozhraní Maps

Mapy (používá se též pojem asociativní pole) ukládají dvojice instancí - klíč a jemu odpovídající hodnotu. Klíče se v mapě nemohou opakovat (obdobně jako v množině). Efektivnější je použití varianty HashMap, varianta TreeMap zajišťuje trvalé setřídění mapy dle klíčů. Rozhraní Iterator umožňuje procházet jednotlivé prvky kolekce bez ohledu na její konkrétní typ (např. když chceme vypsat všechny prvky kolekce). Pokud chceme vkládat instance do struktur, které udržují setříděné prvky (TreeSet a TreeMap), je nutné zajistit porovnání těchto instancí na větší/menší. Toto lze zajistit dvěmi způsoby – buď musí mít vkládané instance implementováno rozhraní Comparable s metodou compareTo nebo při vytvoření dynamické struktury je nutné zadat implementaci rozhraní Comparator, která umí porovnat vkládané objekty.

Třída Collections (neplést s rozhraním Collection) poskytuje další metody pro práci s kontejnery jako je např. třídění, synchronizace datové struktury či „zmražení“ datové struktury.

4.2.2 VYTVOŘENÍ KONTEJNERU

Všechny kontejnery jsou definovány tak, že jejich prvky jsou instance třídy Object a instance potomků třídy Object (což znamená, že do kontejnerů lze vkládat instance libovolných tříd, neboť v Javě jsou všechny třídy potomky třídy Object). Z toho také vyplývá jedno omezení kontejnerů, nelze do nich ukládat primitivní datové typy - pokud je potřebujeme do kontejneru vložit, musíme je převést do objektové podoby, tedy např. celá čísla na instance třídy Integer.

Nejjednodušší způsob vytvoření a vypsání několika kontejnerů je vidět v následujícím příkladě.

PŘÍKLAD 17

```
import java.util.*;
public class VypisKontejneru {
	static Collection napln(Collection c) {
		c.add("pes");
		c.add("pes");
		c.add("kocka");
		return c;
	}

	static Map napln(Map m) {
		m.put("pes", "Alik");
		m.put("pes", "Rek");
		m.put("kocka", "Mina");
		return m;
	}
	public static void main(String[] args) {
		System.out.println(napln(new ArrayList()));
		System.out.println(napln(new HashSet()));
		System.out.println(napln(new HashMap()));
	}
}
```

Výsledkem tohoto programu je následující výpis:

[pes, pes, kocka]
[kocka, pes]

{kocka=Mina, pes=Rek}

Na prvním řádku je výstup z listu, proto se může stejný řetězec opakovat vícekrát. Na druhém řádku je výpis množiny (setu) a tedy každý prvek má jedinečnou hodnotu. Pro výpis mapy na třetím řádku platí totéž, klíč, v tomto případě se řetězec pes, se vyskytuje pouze jednou a to s hodnotou Rek, která byla vkládána jako druhá. Z toho vyplývá, že v případě vložení klíče, který už v mapě je, je původní hodnota přepsána novou. Tedy v našem příkladě Alik je přepsáno řetězcem Rek.

Pokud potřebujeme do kontejneru vložit čísla, tak jednoduché řešení ukazuje následující příklad:

PŘÍKLAD 18

```
ArrayList cisla = new ArrayList();
for(int i = 0; i < 10 i++)
   cisla.add(new Integer(i));
```

Jestliže potřebujeme vytvořit mapu, ve které budou hodnoty také číselné a při každém dalším výskytu stejného klíče se budou měnit, máme dvě možnosti řešení. První je uložit jako hodnotu objektovou reprezentaci čísla a při každé změně jeho hodnoty převádět objekt na jednoduchou proměnnou, provést změnu a znovu převést na objekt (tento způsob je použit v souhrnném příkladě na konci podkapitoly). Druhá možnost řešení je vytvořit si vlastní třídu s proměnnou jednoduchého typu a její instance využít jako hodnoty v mapě, viz následující příklad. V něm se vygeneruje 10000 náhodných celých čísel z intervalu 1 až 20 a zjišťuje se, kolikrát bylo které vygenerováno. Vygenerované číslo je klíčem v mapě (převedené na instance třídy Integer), počty výskytů se počítají pomocí vlastní třídy Counter, jejíž instance jsou hodnoty v mapě.

PŘÍKLAD 19

```
import java.util.*;
class Counter {
        int pocet = 1;
        public String toString() {
            return Integer.toString(pocet);
        }
}
public class Statistika {
        public static void main(String[] args) {
            HashMap hm = new HashMap();
            for(int i = 0; i < 10000; i++) {
                Integer r = new Integer((int)(Math.random() * 20));
                if(hm.containsKey(r))
                    ((Counter)hm.get(r)).pocet++;
                else
                    hm.put(r, new Counter());
            }
            System.out.println(hm);
        }
}
```

Všimněte si, že ve třídě Counter je definována metoda toString, která se automaticky použije pro výpis hodnoty konkrétní instance v metodě System.out.println.

4.2.3 PROCHÁZENÍ KONTEJNERU - ITERÁTOR

Pro postupné procházení jednotlivých seznamů se používají metody rozhraní Iterator. Rozhraní Collection poskytuje metodu iterator(), která vytvoří instanci iterátoru. Pomocí metody next() vrací Iterator jednotlivé hodnoty (nejdříve první, poté postupně následující). Metoda hasNext() zjišťuje, zda následuje další prvek.

Pro následující příklady této kapitoly si definujeme dvě jednoduché třídy:

PŘÍKLAD 20

```
public class Kocka {
	private int cisloKocky;
	Kocka(int i) { cisloKocky = i; }
	void tisk() {
		System.out.println("Kocka #" + cisloKocky);
	}
}

public class Pes {
	private int cisloPsa;
	Pes(int i) { cisloPsa = i; }
	void tisk() {
		System.out.println("Pes #" + cisloPsa);
	}
}
```

V následujícím příkladě si ukážeme, jak vytvořit a naplnit ArrayList a jak jeho obsah vypsat pomocí iterátoru.

PŘÍKLAD 21

```
import java.util.*;
public class Kocky {
	public static void main(String[] args) {
		Collection kocky = new ArrayList();
		for(int i = 0; i < 7; i++)
			kocky.add(new Kocka(i));
		Iterator it = kocky.iterator();
		while(it.hasNext())
			((Kocka)it.next()).tisk();
	}
}
```

Všimněme si deklarace kolekce kocky – pokud nepotřebujeme v programu specifické vlastnosti konkrétní kolekce, obvykle se deklaruje jako typ Collection. Tento postup umožňuje jednoduchou výměnu typu kolekce, pokud by nevyhovovala.

Při výpisu je použito přetypování (Kocka)it.next(), protože jinak není možné použít metodu tisk()třídy Kocka. Po vyjmutí z kolekce nebo mapy je prvek instancí třídy Object a teprve přetypováním zajistíme přiřazení vyjmuté instance ke správné třídě. V tomto se ale může skrývat mnoho chyb, které se projeví až při spuštění programu viz. následující příklad.

PŘÍKLAD 22

```
import java.util.*;
public class KockyAPsi {
	public static void main(String[] args) {
		List kocky = new ArrayList();
		for(int i = 0; i < 7; i++)
			kocky.add(new Kocka(i));
		//není problem pridat do listu i psa (i on je
		//potomkem tridy Object)
		kocky.add(new Pes(7));
		for(int i = 0; i < kocky.size(); i++)
			((Kocka)kocky.get(i)).tisk();
	}
}
```

Na rozdíl od minulého příkladu, kdy pro procházení seznamu byl použit Iterator, zde byla pro procházení seznamu použita metoda get(int i), která vrací hodnotu prvku na zadané pozici v seznamu (a je dostupná pouze pro seznamy – List).

Instanci třídy lze přetypovat na instanci kteréhokoli z jeho předků nikoli na libovolný typ. Nelze tedy z instance třídy Pes udělat instanci třídy Kocka – pokud se o to pokusíte (toto nastane ve výše uvedeném příkladu), vznikne výjimka ClassCastException (více o výjimkách v Kapitole 10).

Java nám poskytuje operátor instanceOf, pomocí kterého lze zjistit, zda instance je instancí zadané třídy. V následujícím příkladě použijeme tento operátor k rozdělení instancí vyjmutých z listu kocky, do kterého uložíme instance tříd Kocka a Pes.

PŘÍKLAD 23

```
import java.util.*;
public class KockyAPsi2 {
	Collection zvirata = new ArrayList();
	KockyAPsi2() {
		for (int i =1;i<11;i++) {
			zvirata.add(new Kocka(i));
			zvirata.add(new Pes(i));
		}
	}
	void vypis(){
		Iterator it = zvirata.iterator();
		Object o;
		while (it.hasNext()) {
			if ((o=it.next()) instanceof Kocka) ((Kocka)o).tisk();
			else ((Pes)o).tisk();
		}
	}
	public static void main (String [] args) {
		KockyAPsi2 kockyAPsi = new KockyAPsi2();
		kockyAPsi.vypis();
	}
}
```

Pro výpis hodnot z mapy nelze přímo použít iterátor, ale je nutné převést mapu na kolekci. Rozhraní Map má k tomuto účelu tři metody – metodu value(), která vytvoří kolekci hodnot, metodu keySet(), která vytvoří množinu klíčů a metodu

entrySet(), pomocí které lze vytvořit z mapy množinu objektů MapEntry obsahujících klíč i příslušnou hodnotu. Pomocí metod getKey() a getValue() pak můžeme získat jednotlivé prvky mapy. Následující metoda vypisMapy (TreeMap tm) ukazuje možné použití těchto metod pro vypsání obsahu mapy.

PŘÍKLAD 24

```
void vypisMapy (TreeMap tm) {
   Iterator it = tm.entrySet().iterator();
   while(it.hasNext()) {
      Map.Entry a = (Map.Entry)it.next();
      System.out.println(a.getKey()+"/t/t"+a.getValue);
   }
}
```

4.2.4 SOUHRNNÝ PŘÍKLAD

Zadání příkladu:

Máme soubor Zvirata.txt:
pes Rek
kocka Micka
kocka Mourek
pes Alik
morce Smudla
morce Fousek
kocka Packa
pes Bety
pes Asta
kocka Paty
pes Fik

Našim úkolem je zjistit, kolik je v souboru záznamů o jednotlivých druzích zvířat. Z každé věty tedy potřebujeme jen její první část, proto použijeme StringTokenizer. Při tvorbě mapy budeme postupovat následovně: pomocí metody containsKey() zjistíme, zda už je tato hodnota klíče v mapě uložena. Pokud ano, pomocí metody get() získáme hodnotu odpovídající tomuto klíči a převedeme ji na číslo, přičteme 1 a hodnotu znovu uložíme do mapy. Pokud se klíč ještě v mapě nevyskytuje, uložíme ho do ní s hodnotou Integer(1). Výsledky pomocí iterátoru vypíšeme.

PŘÍKLAD 25

```
import java.util.*;
import java.io.*;
public class TvorbaMapy {
	public static HashMap vytvor(String jmenoSouboru) {
		LinkedList ln = new LinkedList();
		HashMap tm = new HashMap();
		try {
			BufferedReader veta = new BufferedReader(new
				FileReader(jmenoSouboru));
			String s,s1,s2 = " ";
			while ((s = veta.readLine()) != null){
				StringTokenizer t = new StringTokenizer(s);
				s1 = t.nextToken();
				s2= t.nextToken();
				if (tm.containsKey(s1)) {
					int i = ((Integer)tm.get(s1)).intValue();
					tm.put(s1,new Integer(++i));
				}
				else {
					tm.put(s1, new Integer(1));
				};
			}
			veta.close();
		}
		catch (FileNotFoundException e1) {
			System.out.println("Soubor nenalezen");
		}
		catch (IOException e2 ) {
			System.out.println("Chyba pri cteni");
		}
		return tm;
	}
	public static void tiskMapy( HashMap m){
		Iterator it = m.entrySet().iterator();
		while(it.hasNext()) {
			Map.Entry a = (Map.Entry)it.next();
			System.out.println(a.getKey()+"\t"+a.getValue());
		}
	}
	public static void main (String []args) {
		tiskMapy(vytvor("zvirata.txt"));
	}
}
```

4.3 KONSTANTY

Konstanty v jazy Java jsou deklarovány s klíčovým slovem final. Hodnotu konstanty již nelze dále měnit. Např. pokud deklarujeme konstantu CISLO jako

final int CISLO=10;

potom příkaz

CISLO=15;

způsobí vyvolání chyby a program nebude možno přeložit. Již jsme se zmiňovali o zásadě, že jména konstant by měla být tvořena pouze velkými písmeny. Pokud chceme použít konstantu vně třídy, ve které vznikly, je nutno jejich deklaraci provést vně všech metod, nastavit ji jako veřejnou, tj. za použití klíčového slova public. Jelikož se bude jednat o proměnnou třídy, nezapomeňme, že musí být společná pro všechny instance, je nutno ji deklarovat jako statickou za použití klíčového slova static. Např. takto:

PŘÍKLAD 26

```
public class Neco
{
   public static final int CISLO=10;
}
```

4.4 KOMENTÁŘE

Tučné části kódu definují komentáře naší „Hello,World!“ aplikace:

PŘÍKLAD 27

```
/**
 * The HelloWorldApp class implements an application that
 * simply prints "Hello World!" to standard output.
 */
class MyNewClass {
   public static void main(String[] args) {
      System.out.println("Hello, World!");
   }
}
```

Komentáře jsou ignorovány kompilátorem, ale jsou užitečné pro jiné programátory a hlavně pro nás. Programovací jazyk Java podporuje 3 typy komentářů:

Tabulka 3: Typy komentářů

/* text */	Kompilátor ignoruje cokoliv od /* do */.
/** dokumentace */	Tyto značky indikují dokumentační komentář *(doc comment)*. Kompilátor s nimi zachází naprosto stejně jako s komentáři typu /* a */. Nástroj javadoc používá dokumentační komentáře při vytváření automatické dokumentace.
// text	Kompilátor ignoruje cokoliv od // do konce řádku.

4.5 VSTUP A VÝSTUP DAT

Při běhu programu musí existovat možnost zadávání vstupních dat (vstupní parametry pro výpočet) i možnost práce s výstupními daty (výsledky výpočtů). Hovoříme o terminálovém vstupu a výstupu. Upozorněme, že standardní formátovaný vstup a výstup nejsou v Javě na rozdíl od jazyka C++ určeny pro

vážnou práci, ale pouze pro výukové účely. Java totiž disponuje grafickým rozhraním umožňujícím provádět vstup/výstup údajů s využitím vizuálních komponent, např. textboxů, seznamů, atd. Práce s terminálovým vstupem je v Javě poněkud komplikovanější než v jazyce C++.

Nástroje pro práci se standardním vstupem a výstupem se nacházejí v balíku java.io. Chceme -li pracovat se standardním vstupem a výstupem, použijeme příkaz

```
import java.io.*;
```

Pro práci se standardním formátovaným vstupem v Javě existují objekty System.in a System.out. Objekt System.in realizuje standardní vstup, nejčastěji čtením hodnot z klávesnice. System.out je objekt, který se používá pro zobrazování výstupních údajů na monitoru.

4.5.1 FORMÁTOVANÝ VÝSTUP

Pro formátovaný výstup je používán objekt System.out. S použitím metody print() můžeme provést vytisknutí řetězce na aktuální pozici v řádku. Pokud se na výstup dostane hodnota jiného datového typu než řetězec, je provedena vynucená konverze na řetězec.

```
int a=10;
int b=20;
System.out.print(a);
System.out.print(b);
System.out.print(Ahoj);
```

Na výstupu se objeví: 1020Ahoj. Na rozdíl od jazyka C++ není možno hodnoty na výstupu nijak formátovat, tj. např. stanovit počet desetinných míst.

Chceme-li zřetězit hodnoty na výstupu, tj. vytisknout v jednom kroku více údajů, použijeme operátor +. Můžeme vzájemně sčítat hrušky s jablky, tj. Jak řetězce, tak i proměnné primitivních datových typů. Použijeme

```
System.out.print (a=+a+ b=+b+ Ahoj);
```

obdržíme na výstupu: a=10 b=20 Ahoj. V řetězci se mohou vyskytnout i *escape sekvence*, uveďme dvě nejčastěji používané:

- znak '\t ' (znaková konstanta) resp. (část řetězce): tabulátor.
- znak ' \n' (znaková konstanta) resp. (část řetězce): přechod na novou řádku.

Příkazy

```
System.out.print (a=+a+\n+b=+b+\n Ahoj);
System.out.print (jak+\t+se+\tmate);
```

vytisknou na obrazovku

```
a=10
b=20
```

Ahoj
jak se mate

Stejného efektu dosáhneme za použití metody println().

```
System.out.println(a=+a);
System.out.println(b+b);
System.out.println(jak+\t+se+\tmate);
```

4.5.2 FORMÁTOVANÝ VSTUP

Práce s formátovaným vstupem v Javě je poněkud složitější než v jazyce C++. Je používán objekt System.in. Při zadávání znaků z klávesnice jsou jednotlivé znaky interpretovány jako bajty zakončené ukončovacím znakem ' '. Je vhodné je proto načíst do pole bajtů.

V následujícím příkladu si ukážeme načtení dat z formátovaného vstupu a jejich konverze na typ int.

```
byte [] pole=new byte [30];
System.in.read(pole);
```

Bajty následně převedeme na řetězec typu String

```
String retezec=new String(pole);
```

a ořízneme ukončovací znaky

```
retezec=retezec.trim();
```

Následně provedeme konverzi String na typ int. Nezapomeňme na explicitní přetypování.

```
int cislo=(int)Long.parseLong(retezec);
```

Obdobným způsobem bychom postupovali i při konverzi na jiné datové typy:

```
double cislo=Double.parseDouble(retezec);
```

Tento kód je vhodné obalit do konstrukce try/catch pro odchycení případné výjimky vzniklé např. tím, že uživatel na vstupu zadá textový řetězec obsahující jeden nebo více znaků, které nejsou číslicemi. S tímto přístupem se setkáme až v části věnované odchytávání výjimek.

4.6 OPERÁTORY

Nyní již víme, jak deklarovat a inicializovat proměnnou. Co by nás možná mohlo zajímat, je, jak s proměnnou něco udělat. Operátory programovacího jazyka Java jsou dobrým místem, kde začít. Operátory jsou speciální symboly, které umožňují provádět určité operace nad jedním, dvěma nebo třemi operandy a poté vrátit výsledek.

Předtím, než si vysvětlíme, jak vypadají operátory programovacího jazyka Java, mohlo vy pro nás být užitečné dopředu vědět, který operátor má větší přednost. Operátory jsou v následující tabulce řazeny podle jejich priority. Čím výše je operátor v tabulce, tím větší má prioritu. Operátory s vyšší prioritou jsou vykonávány dříve než operátory s prioritou nižší. Operátory na stejném řádku mají stejnou prioritu. Když jsou v jednom výrazu za sebou dva operátory stejné priority, uplatňuje se pravidlo dané přesnosti. Všechny binární operátory (operují nad dvěma operandy), vyjma přiřazovacích operátorů, jsou vykonávány zleva doprava, u přiřazovacích operátorů je to naopak.

Tabulka 4: Operátory

Operátory	Přednost
postixové	*výraz*++ *výraz*--
unární	++*výraz* --*výraz* +*výraz* -*výraz* ~ !
multiplikativní	* / %
aditivní	+ -
posunu	<< >> >>>
relační	< > <= >= instanceof
rovnosti	== !=
bitového AND	&
bitového exklusivního OR	^
bitového inklusivního OR	|
logického AND	&&
logického OR	||
ternární	? :
přiřazení	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Obecně se dá říct, že některé operátory se používají o mnoho více než jiné; například operátor přiřazení „=“ je o mnoho více používán než operátor pravého posuvu bez znaménka „>>>“. Podle tohoto pravidla probereme nejdříve obecně nejpoužívanější operátory a kapitolu zakončíme těmi speciálnějšími. Každá podkapitola je zakončena příkladem, který nám pomůže téma lépe pochopit.

4.6.1 JEDNODUCHÝ PŘIŘAZOVACÍ OPERÁTOR

Jeden z nejčastějších operátorů, se kterými se setkáte, je jednoduchý přiřazovací operátor „=“. Tento operátor jsme mohli vidět ve třídě Bicycle; přiřazuje hodnotu pravého operandu levému:

```
int cadence = 0;
int speed = 0;
int gear = 1;
```

Tento operátor může také být použit s objekty k přiřazení objektové reference, o které se budeme bavit v příštích kapitolách.

4.6.2 POČETNÍ OPERÁTORY

Programovací jazyk Java poskytuje operátory, které slouží ke sčítání, odčítání, násobení a dělení operandů. Tyto operátory jsou obdoby svých matematických

protějšků. Jediný operátor, který pro vás může působit nově, je „%“, který vydělí jeden operand druhým a vrátí zbytek jako výsledek.

Tabulka 5: Početní operátory

Operátor	Název
+	Operátor sčítání (taktéž spojení řetězců znaků)
−	Operátor odčítání
*	Operátor násobení
/	Operátor dělení
%	Operátor zbytku po dělení

Následující program testuje početní operátory.

PŘÍKLAD 28

```
class ArithmeticDemo {

    public static void main (String[] args){

        int vysledek = 1 + 2; // výsledek je nyní 3
        System.out.println(vysledek);

        vysledek = vysledek - 1; // výsledek je nyní 2
        System.out.println(vysledek);

        vysledek = vysledek * 2; // výsledek je nyní 4
        System.out.println(vysledek);

        vysledek = vysledek / 2; // výsledek je nyní 2
        System.out.println(vysledek);

        vysledek = vysledek + 8; // výsledek je nyní 10
        vysledek = vysledek % 7; // výsledek je nyní 3
        System.out.println(vysledek);

    }
}
```

Můžeme také kombinovat početní operátory s jednoduchým přiřazovacím operátorem za vzniku složených přiřazení. Například jak x += 1;, tak x = x + 1; zvýší hodnotu x o 1.

Operátor + může být použit pro spojování řetězců, jak je také ukázáno v následujícím programu:

PŘÍKLAD 29

```
class ConcatDemo {
    public static void main(String[] args){
        String prvniRetezec = "Toto je";
        String druhyRetezec = " spojovaný řetězec.";
        String tretiRetezec = prvniRetezec + druhyRetezec;
        System.out.println(tretiRetezec);
    }
        }
```

Na konci tohoto programu obsahuje proměnná thirdString řetězec „Toto je spojovaný řetězec.“, který je taktéž vytištěn na standardní výstup.

4.6.3 UNÁRNÍ OPERÁTORY

Unární operátory potřebují pouze jeden operand; poskytují různé operace jako zvyšování/snižování hodnoty proměnné o jedna, negaci výrazu nebo převracení hodnoty typu boolean.

Tabulka 6: Unární operátory

Operátor	Název	Popis
+	Unární plus	Indikuje kladné číslo (čísla jsou kladná i bez něj)
−	Unární mínus	Neguje výraz
++	Inkrementace	Zvyšuje hodnotu o 1
−−	Dekrementace	Snižuje hodnotu o 1
!	Logický doplněk	Převrací hodnotu typu boolean

Následující program testuje unární operátory:

PŘÍKLAD 30

```
class UnaryDemo {

    public static void main(String[] args){
        int vysledek = +1; // výsledek je nyní 1
        System.out.println(vysledek);
        vysledek--;  // výsledek je nyní 0
        System.out.println(vysledek);
        vysledek++; // výsledek je nyní 1
        System.out.println(vysledek);
        vysledek = -vysledek; // výsledek je nyní -1
        System.out.println(vysledek);
        boolean uspech = false;
        System.out.println(uspech); // false
        System.out.println(!uspech); // true
    }
}
```

Operátory inkrementace/dekrementace mohou být aplikovány **před** (prefix) a **po** (postfix) operandu. Jak kód result++;, tak ++result; vyústí ve zvýšení hodnoty proměnné result o jedna. Jediným rozdílem je, že prefixovaná forma (++result) vrátí inkrementované číslo, zatímco postfixová forma (result++) vrátí původní číslo. Jestliže tento operátor použijeme pouze při jednoduché inkrementaci/dekrementaci, nezáleží na tom, kterou verzi použijeme. Ale pokud budeme tento operátor chtít použít v rámci složitého výrazu, špatná volba může vyústit ve špatný výsledek.

Následující program ilustruje rozdíl mezi prefixovou a postfixovou formou:

PŘÍKLAD 31 ,

```
class PrePostDemo {
    public static void main(String[] args){
        int i = 3;
            i++;
            System.out.println(i);   // "4"
            ++i;
            System.out.println(i);   // "5"
```

```
        System.out.println(++i);          // "6"
        System.out.println(i++);          // "6"
        System.out.println(i);   // "7"
    }
}
```

4.6.4 POROVNÁVACÍ OPERÁTORY

Operátory rovnosti a porovnávací operátory umožňují operace, jako jsou: větší než, menší než, je roven nebo není roven. Většina těchto operátorů nám bude připadat známá. Mysleme však na to, že **musíme použít „==“, ne „=“, pokud chceme testovat, zda jsou si dvě primitivní hodnoty rovny**.

Tabulka 7: Porovnávací operátory

Operátor	Název
==	je rovno
!=	není rovno
>	větší než
>=	větší nebo rovno
<	menší než
<=	menší nebo rovno

Následující program testuje operátory porovnání:

PŘÍKLAD 32

```
class ComparisonDemo {

    public static void main(String[] args){
        int hodnota1 = 1;
        int hodnota2 = 2;
        if(hodnota1 == hodnota2) System.out.println("hodnota1 == hodnota2");
        if(hodnota1 != hodnota2) System.out.println("hodnota1 != hodnota2");
        if(hodnota1 > hodnota2) System.out.println("hodnota1 > hodnota2");
        if(hodnota1 < hodnota2) System.out.println("hodnota1 < hodnota2");
        if(hodnota1 <= hodnota2) System.out.println("hodnota1 <= hodnota2");
    }
}
```

Výstup:

```
hodnota1 != hodnota2
hodnota1 < hodnota2
hodnota1 <= hodnota2
```

4.6.5 PODMÍNKOVÉ OPERÁTORY

Operátory && a || poskytují operace logického AND (a zároveň) a logického OR (nebo) nad dvěma výrazy typu boolean. Tyto operátory používají zkrácené vyhodnocování, což znamená, že druhá operace se vykoná pouze v případě, že je to potřeba.

Tabulka 8: Podmínkové operátory

Operátor	Název

&&	logické AND (a zároveň)
||	logické OR (nebo)

Následující program testuje tyto operátory:

PŘÍKLAD 33

```
class ConditionalDemo1 {

    public static void main(String[] args){
        int hodnota1 = 1;
        int hodnota2 = 2;
        if((hodnota1 == 1) && (hodnota2 == 2)) System.out.println("hodnota1 je 1 AND hodnota2 je 2");
        if((hodnota1 == 1) || (hodnota2 == 1)) System.out.println("hodnota1 je 1 OR hodnota2 je 1");

    }
}
```

Dalším podmínkovým operátorem je ?:, který může být chápan jako zkratka za příkaz if-then-else, o kterém se budeme bavit později. Tento operátor je taktéž znám jako **ternární operátor**, protože pracuje nad třemi operandy. V následujícím příkladu může být tento operátor přečten jako „Jestliže nejakaPodminka je true, přiřaď hodnotu proměnné hodnota1 do vysledek. Pokud ne, přiřaď hodnotu proměnné hodnota2 do vysledek.“ Následující program tento operátor testuje:

PŘÍKLAD 34

```
class ConditionalDemo2 {

    public static void main(String[] args){
        int hodnota1 = 1;
        int hodnota2 = 2;
        int vysledek;
        boolean nejakaPodminka = true;
        vysledek = nejakaPodminka ? hodnota1 : hodnota2;

        System.out.println(vysledek);

    }
}
```

Protože nejakaPromenna je true, program vypíše na obrazovku „1“. Tento operátor je vhodné v jednodušších případech používat pro zpřehlednění kódu místo konstrukce if-then-else;.

4.7 VÝRAZY, PŘÍKAZY A BLOKY

Nyní, když rozumíme proměnným a operátorům, je čas popovídat si o **výrazech**, **příkazech** a **blocích**. Operátory mohou být použity ve výrazech, které vracejí hodnotu; výrazy jsou základní součástí příkazů; příkazy mohou být seskupeny do bloků.

4.7.1 VÝRAZY

Výraz je konstrukce složená z proměnných, operátorů a volání metod, které jsou složeny pomocí odpovídající syntaxe jazyka a vyúsťují v jednu hodnotu. Již jsme mohli vidět příklady výrazů, zde jsou vyznačeny podtržením:

PŘÍKLAD 35

```
int cadence = 0;
nejakePole[0] = 100;
System.out.println("Element 1 na indexu 0: " + nejakePole[0]);
int vysledek = 1 + 2; // výsledek je nyní 3
if(hodnota1 == hodnota2) System.out.println("hodnota1 == hodnota2");
```

Datový typ hodnoty vrácené výrazem záleží na tom, z čeho se výraz skládá. Výraz cadence = 0 vrátí int, protože přiřazovací operátor vrací hodnotu takového typu, jaký je na jeho levé straně; v tomto případě cadence je int. Jak můžeme vidět na jiných příkladech, výraz může stejně dobře vracet jakýkoliv jiný typ, jako je boolean nebo string.

Programovací jazyk Java nám dovolí konstruovat složené výrazy z různých menších výrazů, dokud typ dat, požadovaný jednou částí výrazu, neodpovídá typu dat dalšího. Zde je příklad složeného výrazu:

```
1 * 2 * 3
```

V tomto speciálním případě je pořadí, v jakém se výraz vykonává nedůležité, protože výsledek násobení nezávisí na pořadí; výsledek je vždy stejný, lhostejno v jakém pořadí je násobení vykonáváno. Nicméně toto neplatí pro všechny případy. Můžeme si uvést příklad, který vrací různé výsledky podle toho, zda se provede dřív sčítání nebo dělení:

```
x + y / 100     // dvojsmyslné
```

Můžeme přesně specifikovat, která operace se provede dříve použitím kulatých závorek (a). Například, abychom udělali předchozí příklad jednoznačný, stačí provést tuto změnu:

```
(x + y) / 100    // jednoznačné, doporučované
```

Pokud přesně nespecifikujeme, v jakém pořadí se bude výraz vyhodnocovat, odvodí se podle priority operátorů, použitých ve výrazu. Operátor, který má vyšší přednost, se provede dříve. Například operátor dělení má větší prioritu než operátor sčítání. Právě proto jsou následující výrazy shodné:

```
x + y / 100
x + (y / 100) // jednoznačné, doporučované
```

Když píšeme složité výrazy, je dobré přesně označit, která část se má provést dříve. Tato praktika činí kód lépe čitelným a udržovatelným.

4.7.2 PŘÍKAZY

Příkazy jsou zhruba shodné s větami běžných jazyků. Příkaz tvoří kompletní jednotku vykonávání kódu. Následující typy výrazů mohou být použity jako příkazy pouhým ukončením středníkem (;).

- přiřazovací výrazy
- jakékoliv použití ++ nebo --
- volání metod
- tvoření objektů

Tyto příkazy jsou nazývány **výrazové příkazy**. Zde je několik příkladů výrazových příkazů:

```
nejakaHodnota = 8933.234;          // příkaz přiřazení
nejakaHodnota++;                   // příkaz inkrementace
System.out.println("Hello World!");// příkaz volání metody
Bicycle mojeKolo = new Bicycle();  // příkaz vytvoření objektu
```

Společně s výrazovými příkazy jsou zde další dva druhy příkazů: **deklarativní příkazy** a **příkazy toku programu**. Deklarativní příkazy deklarují proměnnou. Doteď jste viděli spoustu příkladů deklarativních příkazů:

```
double nejakaHodnota = 8933.234; // deklarativní příkaz
```

Konečně, příkazy toku programu regulují, v jakém pořadí budou jednotlivé příkazy vykonávány. Více se o nich dovíme dále v této kapitole.

4.8 PŘÍKAZY TOKU PROGRAMU

Příkazy jsou uvnitř zdrojového kódu vykonávány od prvního směrem dolů tak, jak jsou v kódu napsány. Příkazy toku programu nicméně zastaví tok a umožňují náš program řídit, opakovat a větvit, umožňují našemu programu podmíněně vykonávat příslušné bloky kódu. Tato sekce popisuje rozhodovací příkazy (if-then, if-then-else, switch), příkazy smyčky (for, while, do-while) a příkazy přesunu (break, continue, return) podporované programovacím jazykem Java.

4.8.1 PŘÍKAZY IF-THEN A IF-THEN-ELSE

Příkaz if-then

Příkaz if-then je nejzákladnějším ze všech příkazů toku programu. Říká, aby část kódu byla vykonána pouze tehdy, když se příslušný test vyhodnotí jako true (pravda). Například třída Bicycle může povolit použití brzd pouze tehdy, když je kolo již v pohybu. Jedna z možných implementací metody applyBrakes může vypadat takto:

PŘÍKLAD 36

```
void applyBrakes(){
      if (jeKoloVPohybu){ // část "if" (když): kolo se musí pohybovat
     momentalniRychlost--;  // část "then" (tak): sniž současnou
```

```
                    rychlost
    }
}
```

Jestliže se podmínka vyhodnotí jako false (což znamená, že kolo není v pohybu), program přeskočí na konec příkazu if-then.

Dále, složené závorky jsou dobrovolné, pokud část „then“ obsahuje pouze jeden příkaz:

PŘÍKLAD 37

```
void applyBrakes(){
    if (jeKoloVPohybu) momentalniRychlost--; // stejné jako výše, ale bez složených závorek
}
```

Vynechávání závorek je otázkou našeho zvyku, činí to však kód více křehčím. Pokud v budoucnu budeme chtít do části „then“ přidat nový příkaz, můžeme zapomenout nově potřebné složené závorky. Kompilátor však tuto chybu nemůže odhalit, podle něj je takový kód správný.

Příkaz if-then-else

Příkaz if-then-else poskytuje druhou cestu vykonávání programu, když se část „if“ vyhodnotí jako false. Můžeme použít příkaz if-then-else v metodě applyBrakes pro provedení nějaké akce, když kolo není v pohybu. V tomto případě jednoduše vytiskneme chybovou zprávu, pokud kolo není v pohybu.

PŘÍKLAD 38

```
void applyBrakes(){
    if (jeKoloVPohybu) {
        momentalniRychlost--;
    } else {
        System.err.println("Kolo již stojí!");
    }
}
```

Následující program dává školní známku, založenou podle výsledku testu: A pro skóre nad 90 %, B pro skóre nad 80 % atd.:

PŘÍKLAD 39

```
class IfElseDemo {
   public static void main(String[] args) {

       int testscore = 76;
       char znamka;

       if (testscore >= 90) {
           znamka = 'A';
       } else if (testscore >= 80) {
           znamka = 'B';
       } else if (testscore >= 70) {
           znamka = 'C';
       } else if (testscore >= 60) {
```

```
            znamka = 'D';
        } else {
            znamka = 'F';
        }
        System.out.println("Vase znamka je " + znamka);
    }
}
```

Výstup tohoto programu je:

Vase znamka je C

Jak si můžeme všimnout, hodnota testscore může vyhovět více než jedné podmínce ve složeném příkaze: 76 >= 70 a 76 >= 60. Nicméně, jakmile je jednou podmínka splněna, odpovídající úsek programu je vykonán (znamka = 'C';) a zbytek podmínek není vyhodnocen.

Příkaz switch

Na rozdíl od if-then a if-then-else, příkaz switch povoluje libovolný počet cest, kterými se program vydá. switch pracuje s těmito primitivními typy: byte, short, char a int. Taktéž pracuje s výčtovými typy a se speciálními třídami obalujícími primitivní datové typy: Character, Byte, Short a Integer. O všech těchto záležitostech se je možno si přečíst v [16] .

Následující program definuje hodnotu typu int nazvanou mesic, jejíž hodnota reprezentuje aktuální měsíc. Následující program vypíše název měsíce s použitím příkazu switch.

PŘÍKLAD 40

```
class SwitchDemo {
    public static void main(String[] args) {

        int mesic = 8;
        switch (mesic) {
            case 1:  System.out.println("leden"); break;
            case 2:  System.out.println("únor"); break;
            case 3:  System.out.println("březen"); break;
            case 4:  System.out.println("duben"); break;
            case 5:  System.out.println("květen"); break;
            case 6:  System.out.println("červen"); break;
            case 7:  System.out.println("červenec"); break;
            case 8:  System.out.println("srpen"); break;
            case 9:  System.out.println("září"); break;
            case 10: System.out.println("říjen"); break;
            case 11: System.out.println("listopad"); break;
            case 12: System.out.println("prosinec"); break;
            default: System.out.println("špatný měsíc.");break;
        }
    }
}
```

V tomto případě je na standardní výstup vytištěno: Srpen.

Tělo příkazu switch je známo jako *switch blok*. Jakýkoli příkaz, který je ve switch bloku, by měl být označen pomocí slov case nebo default. Příkaz switch vyhodnotí svůj výraz a po porovnání vykoná odpovídající blok označený slovem case. Samozřejmě totéž můžeme uskutečnit pomocí příkazů if-then-else:

PŘÍKLAD 41

```
int mesic = 8;
if (mesic == 1) {
   System.out.println("Leden");
} else if (mesic == 2) {
   System.out.println("Únor");
}
. . . // a tak dále
```

Rozhodnutí, kdy použijete if-then-else a kdy switch, záleží na našem mínění. Měli bychom se opřít o čitelnost kódu a o obdobné faktory. Příkaz if-then-else může být použit k rozhodování nad několika hodnotami a složitými podmínkami, zatímco příkaz switch může být použit k rozhodování pouze nad jednoduchou celočíselnou nebo výčtovou hodnotou.

Nyní se věnujme příkazu break, který následuje po každém case. Každý příkaz break ukončí provádění daného příkazu switch. Tok programu pokračuje po ukončovací složené závorce *switch bloku*. Příkazy break jsou nezbytné, protože bez nich by každý příkaz case tzv. „propadl“ - to znamená, že bez přímého zadání příkazu break tok programu bude pokračovat skrz další příkazy case, tudíž by byly provedeny i bloky, které provedy být nemají. Následující program ukazuje, kdy nám může být užitečné, že příkazy case propadají:

PŘÍKLAD 42

```
class SwitchDemo2 {
   public static void main(String[] args) {

      int mesic = 2;
      int rok = 2000;
      int pocetDni = 0;

      switch (mesic) {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            pocetDni = 31;
            break;
         case 4:
         case 6:
         case 9:
         case 11:
            pocetDni = 30;
            break;
         case 2:
            if ( ((rok % 4 == 0) && !(rok % 100 == 0))
                 || (rok % 400 == 0) )
               pocetDni = 29;
            else
               pocetDni = 28;
            break;
         default:
            System.out.println("Špatný měsíc.");
            break;
      }
      System.out.println("Počet dnů v měsíci = " + pocetDni);
   }
}
```

Zde je výstup programu.

Počet dnů v měsíci = 29

Technicky není poslední příkaz break ve *switch bloku* nutný, protože tok programu přejde za příkaz switch i bez něj. Nicméně je vhodné používat příkaz break i na konci, protože to činí náš kód lépe modifikovatelným a méně náchylný k chybám. Sekce default zachytává ten případ, kdy není žádná sekce case vykonána.

Příkazy while *a* do-while

Příkaz while (nebo také smyčka while) opakovaně vykonává kód, dokud je podmínkový výraz pravdivý (true).

Jeho syntaxe může být zhrnuta takto:

while (*podmínkovýVýraz*) {
    příkaz(y)
}

Příkaz while vyhodnotí podmínkovýVýraz, který musí vracet hodnotu typu boolean. Jestliže je jeho hodnota true, příkaz while vykoná příkaz(y) v bloku while. Příkaz while pokračuje vyhodnocováním podmínkového výrazu a vykonáváním bloku, dokud podmínka nevrátí false. Použití příkazu while k vypsání hodnot od jedné do desíti může být uděláno stejně jako v následujícím programu:

PŘÍKLAD 43

```
class WhileDemo {
    public static void main(String[] args){
        int cislo = 1;
        while (cislo < 11) {
            System.out.println("Číslo je nyní: " + cislo);
            cislo++;
        }
    }
}
```

Můžeme také udělat nekonečnou smyčku použitím následujícího úryvku kódu:

```
while (true){
    // zde bude váš kód
}
```

Programovací jazyk Java taktéž poskytuje smyčku do-while, která se dá shrnout takto:

```
do {
    příkaz(y)
} while (podmínkovýVýraz);
```

Rozdíl mezi do-while a while je, že smyčka do-while vykoná svůj podmínkový výraz až po vykonání příkazů smyčky. Proto jsou příkazy v bloku do vždy vykonány alespoň jednou, jak ukazuje následující program:

PŘÍKLAD 44

```
class DoWhileDemo {
    public static void main(String[] args){
        public static void main(String[] args){
          int cislo = 1;
        do {
                    System.out.println("Číslo je nyní: " + cislo);
                    cislo++;
          } while (cislo < 11);
          }
    }
}
```

Příkaz for

Příkaz for poskytuje komplexní způsob, jak projít přes řadu hodnot. Programátoři o tomto příkazu často mluví jako o „cyklu for“, protože je to cesta, jak

opakovaně vykonávat kód, dokud není splněna daná podmínka. Obecná forma příkazu for může být vyjádřena jako v následující ukázce:

```
for (inicializace; podmínka; inkrementace) {
    příkaz(y)
}
```

Když používáme tuto verzi příkazu for, mějme na paměti, že:

- Výraz *inicializace* připraví smyčku; je vykonán pouze jednou, když smyčka začíná.
- Když výraz *podmínky* vrátí hodnotu false, smyčka končí.
- Výraz *inkrementace* je vykonán po každém průchodu smyčkou; v tomto místě lze provádět inkrementaci (zvětšení) nebo dekrementaci (zmenšení) hodnoty.

Následující program ukazuje příklad, kdy je smyčka for použita k výpisu čísel od 1 do 10:

PŘÍKLAD 45

```
class ForDemo {
    public static void main(String[] args){
        for(int i=1; i<11; i++){
            System.out.println("Cislo je: " + i);
        }
    }
}
```

Výstup tohoto programu je:

```
Cislo je: 1
Cislo je: 2
Cislo je: 3
Cislo je: 4
Cislo je: 5
Cislo je: 6
Cislo je: 7
Cislo je: 8
Cislo je: 9
Cislo je: 10
```

Všimněme si, jak kód deklaruje proměnou uvnitř inicializačního výrazu. Viditelnost proměnné je od své deklarace po konec bloku for, takže může být použita i uvnitř výrazů podmínky a inkrementace. Jestliže proměnná, která kontroluje příkaz for, není potřeba mimo smyčku, je nejlepší místo k její deklaraci v inicializačním výrazu. Pro kontrolu smyčky for se často používají proměnné s názvy i, j a k; jejich deklarováním uvnitř inicializačního výrazu omezujeme jejich životnost a zabraňujeme chybám.

Není třeba vždy uvádět všechny 3 výrazy smyčky for; nekonečná smyčka může být vytvořena např. takto:

```
for ( ; ; ) {    // nekonečná smyčka
```

```
    // zde bude váš kód
}
```

Příkaz for má taktéž jinou formu, určenou pro procházení polí a kolekcí.

Tato forma je někdy nazývána rozšířený for a může náš kód zkrátit a udělat ho lépe čitelným. Pro demonstraci použijeme následující pole celých čísel, obsahující hodnoty od 1 do 10:

```
int[] cisla = {1,2,3,4,5,6,7,8,9,10};
```

Následující program používá rozšířenou formu smyčky for pro průchod polem:

PŘÍKLAD 46

```
class EnhancedForDemo {
    public static void main(String[] args){
        int[] cisla = {1,2,3,4,5,6,7,8,9,10};
        for (int cislo : cisla) {
          System.out.println("Cislo je: " + cislo);
        }
    }
}
```

V tomto příkladě proměnná cislo nabývá postupně vždy jednu z hodnot z pole čísel. Výstup programu je stejný jako předtím:

```
Cislo je: 1
Cislo je: 2
Cislo je: 3
Cislo je: 4
Cislo je: 5
Cislo je: 6
Cislo je: 7
Cislo je: 8
Cislo je: 9
Cislo je: 10
```

Je vhodné používat tuto formu příkazu for místo té základní, kdekoli to jen jde.

4.8.2 PŘÍKAZY PRO OVLIVNĚNÍ PRŮBĚHU CYKLU

Příkaz break

Příkaz break má dvě formy: **označenou** a **neoznačenou**. Neoznačenou formu jsme mohli vidět, když jsme se bavili o příkazu switch.

Neoznačený příkaz break můžeme použít k násilnému zastavení příkazů for, while nebo do-while, jak ukazuje následující program:

PŘÍKLAD 47

```
class BreakDemo {
   public static void main(String[] args) {
```

```
        int[] poleCelychCisel = { 32, 87, 3, 589, 12, 1076,
                                  2000, 8, 622, 127 };
        int hledane = 12;

        int i;
        boolean nalezeno = false;

        for (i = 0; i < poleCelychCisel.length; i++) {
            if (poleCelychCisel[i] == hledane) {
                nalezeno = true;
                break;
            }
        }

        if (nalezeno) {
            System.out.println("Cislo " + hledane
                               + " nalezeno na indexu " + i);
        } else {
            System.out.println(hledane
                               + " neni v poli");
        }
    }
}
```

Tento program hledá číslo 12 v poli poleCelychCisel. Příkaz break, který je zvýrazněn tučně, ukončí smyčku for, když je hodnota nalezena. Tok programu se poté přesune na výpis hlášení a ukončí se.

Výstup programu je:

```
Cislo 12 nalezeno na indexu 4
```

Neoznačený příkaz break ukončí nejbližší (ve smyslu nejvíce „vnitřní“) příkaz switch, for, while nebo do-while, ale označený break ukončí i vnější. Následující program je podobný předchozímu, ale používá vnořené smyčky for pro nalezení hodnoty ve dvourozměrném poli. Když je hodnota nalezena, označený break ukončí vnější smyčku for, která je označena návěštím hledani:

PŘÍKLAD 48

```
class BreakWithLabelDemo {
    public static void main(String[] args) {

        int[][] poleCelychCisel = { { 32, 87, 3, 589 },
                                    { 12, 1076, 2000, 8 },
                                    { 622, 127, 77, 955 }
                                  };
        int hledane = 12;

        int i;
        int j = 0;
        boolean nalezeno = false;

    hledani:
        for (i = 0; i < poleCelychCisel.length; i++) {
            for (j = 0; j < poleCelychCisel[i].length; j++) {
                if (poleCelychCisel[i][j] == hledane) {
                    nalezeno = true;
                    break hledani;
                }
            }
        }

        if (nalezeno) {
            System.out.println("Cislo " + searchfor +
                               " nalezeno na indexu " + i + ", " + j);
        } else {
            System.out.println(searchfor
                               + " neni v poli");
        }
    }
}
```

Zde je výstup programu:

Cislo 12 nalezeno na indexu 1, 0

Příkaz break ukončil příkaz označený návěštím hledani; nepřenesl však další tok programu na místo označené tímto návěštím, nýbrž na **první příkaz za koncem návěštím označeného bloku** – tedy na řádek „if (nalezeno) {“.

Příkaz continue

Příkaz continue ukončí aktuální iteraci smyčky for, while nebo do-while. Neoznačená forma příkazu přeskočí na konec bloku smyčky a pokračuje vyhodnocením výrazu podmínky. Následující program prochází přes řetězec prohledejMe a počítá počet výskytů písmene „p“. Jestliže aktuální písmenko není „p“, příkaz continue přeskočí zbytek smyčky a pokračuje dalším znakem. Jestliže to je „p“, program zvýší počet výskytů.

PŘÍKLAD 49

```
class ContinueDemo {
    public static void main(String[] args) {

        String prohledejMe = "peter piper picked a peck of pickled peppers";
        int max = prohledejMe.length();
        int pocetPecek = 0;

        for (int i = 0; i < max; i++) {
            // zajímá nás pouze p
            if (prohledejMe.charAt(i) != 'p')
                continue;

            // zpracuj p
            pocetPecek++;
        }
        System.out.println("Nalezeno " + pocetPecek + " výskytů písmene p v řetězci.");
    }
}
```

Zde je výstup programu:

Nalezeno 9 výskytů písmené p v řetězci.

Abychom viděli efekt příkazu continue, odstraňme jej a příkaz překompilujme. Když program spustíme, zjistíme, že pracuje špatně, protože nalezl 35 „p“ místo 9.

Označený příkaz continue přeskočí zbytek iterace označené smyčky. Následující program používá dvě vložené smyčky k nalezení podřetězce v řetězci. Dvě vložené smyčky jsou potřeba: jedna iteruje přes hledaný podřetězec a jedna přes prohledávaný řetězec.

PŘÍKLAD 50

```
class ContinueWithLabelDemo {
    public static void main(String[] args) {

        String prohledejMe = "Look for a substring in me";
        String podretezec = "sub";
        boolean nalezeno = false;

        int max = prohledejMe.length() - podretezec.length();

    test:
        for (int i = 0; i <= max; i++) {
            int n = podretezec.length();
            int j = i;
            int k = 0;
            while (n-- != 0) {
                if (prohledejMe.charAt(j++) != podretezec.charAt(k++)) {
                    continue test;
                }
            }
            nalezeno = true;
                break test;
        }
        System.out.println(nalezeno ? "Nalezeno" :
                                      "Nenalezeno");
    }
}
```

Zde je výstup tohoto programu:

Nalezeno

Příkaz return

Příkaz return ukončí právě vykonávanou metodu a tok programu se vrátí na místo, kde byla metoda zavolána. Příkaz return má dvě formy: jedna vrací hodnotu a jedna ne. Pro vrácení hodnoty jednoduše napíšeme hodnotu (nebo výraz, který ji spočte) za klíčové slovo return.

return ++count;

Typ vrácených dat musí být shodný s typem deklarovaným v hlavičce metody. Když je metoda deklarována s klíčovým slovem void, použijeme formu příkazu return, která hodnotu nevrací.

return;

5 METODY

Metoda patří mezi nejčastěji používané nástroje (téměř) každého programovacího jazyka. Představuje samostatnou část programu, konající nějakou specializovanou funkci. Metody jsou umístěny mimo hlavní program, mohou být spouštěny z main()nebo z jakékoliv jiné metody. Metodu zpravidla voláme se seznamem argumentů, kterým předáváme hodnoty potřebné pro výpočet. Metoda většinou vrací nějaký výsledek. Existují i metody, kterým nepředáváme žádné údaje, nebo naopak žádné výsledky nevrací. Výhodou jejich používání je zjednodušení struktury programu či možnost opakovaného provádění výpočtů (tj. nemusíme psát znovu celý kód, pouze zavoláme příslušnou metodu). Z hlediska OOP lze metody chápat jako **zprávy** zaslané instancím nebo třídám. Metody bývají někdy označovány jako funkce či podprogramy, ale toto označení není přesné.

Metody dělíme do dvou skupin:

- metody třídy (statické metody),
- metody instance.

Nejprve se seznámíme s první skupinou metod. **Metody třídy** představují zprávy zaslané třídě jako celku, nelze je volat pro konkrétní instanci. Mohou být používány v případech, kdy ještě neexistuje žádná instance třídy. Metody třídy bývají označovány jako **statické metody**. Ve statických metodách lze používat pouze statické proměnné nebo lokální proměnné, nemohou v nich být používány proměnné instance. Statické metody jsou v Javě používány při matematických výpočtech. Například Math.sin(x) představuje statickou metodu třídy Math.

5.1 DEKLARACE A VOLÁNÍ METODY

Deklarace metody je tvořena hlavičkou metody a tělem metody. Hlavička obsahuje informaci o jménu metody, typu návratové hodnoty a seznam argumentů. Jméno metody by mělo vyjadřovat činnost metody. Tělo metody tvoří výkonný kód umístěný uvnitř složených závorek. Metody třídy jsou deklarovány s klíčovým slovem static. Uveďme, že na rozdíl od jazyka C++ nemusíme vytvářet prototyp funkce. V Javě není podstatné, zda volání metody předchází její deklaraci či naopak. Podívejme se poněkud podrobněji na problematiku parametrů metod. Často se setkáváme s pojmy formální parametry metod a skutečné parametry metod. **Formální parametry** nalezneme v deklaraci metody v její hlavičce. Stejně jako jakékoliv jiné proměnné musí být i formální parametry deklarovány s uvedením datového typu a názvu. **Skutečné parametry** používáme při volání metody, s jejich hodnotami metodu voláme. Skutečné parametry by měly být stejného datového typu jako parametry formální. Nejsou-li, je provedena implicitní konverze (pokud je to možné).

Předávání hodnot skutečných parametrů parametrům formálním je prováděno při volání metody. Hodnoty jsou předávány postupně, tj. hodnotě prvního formálního parametru je předána hodnota prvního skutečného parametru, hodnotě druhého formálního parametru hodnota druhého skutečného parametru, atd.

V Javě jsou předávány parametry primitivních datových typů hodnotou. Formální parametr představuje kopii skutečného parametru. Jakákoliv změna hodnoty formálního parametru neovlivní hodnotu skutečného parametru. Pokud metoda nevrací žádnou hodnotu, je její návratový typ void. Má-li metoda návratovou hodnotu (tj. její návratová hodnota je jiného typu než void), musí metoda obsahovat

klíčové slovo return s uvedením hodnoty či výrazu, který bude metoda navracet jako výsledek. Dále již nesmí následovat žádný kód. Byl by nedostupný, tzv. unreachable. Metoda v Javě je schopna vrátit nejvýše jednu hodnotu. Nelze tedy vytvořit metodu, která by přímo vracela dvě hodnoty. Toto omezení lze obejít předáváním parametrů odkazem (viz dále).

Podívejme se na následující příklad. Metoda obvod()slouží k výpočtu obvodu kruhu.

PŘÍKLAD 51

```
static double obvod (double r)// formalni parametr r
{
        return 2*Math.PI*r;
}
```

Pokud je tělo metody krátké, lze metodu zapsat do jednoho řádku.

```
static double obvod (double r){return 2*Math.PI*r;}
```

Metoda má jeden formální parametr typu double, vrací také hodnotu typu double. Za klíčovým slovem return následuje výpočet 2r. Metodu voláme z hlavního programu uvedením jejího názvu a seznamu skutečných parametrů v kulatých závorkách.

PŘÍKLAD 52

```
public class kruh
{
//deklarace metody obvod
static double obvod (double r)
{
        return 2*Math.PI*r;
}
//deklarace metody main
public static void main (String [] args)
{
        double pol=50;
        // volani metody obvod se skutecnym parametrem pol
        double obv=obvod(pol);
        System.out.println(obv);
}
}
```

V Javě existuje možnost předávání **parametrů odkazem**. Týká se pouze neprimitivních datových typů, tj. polí a objektů. V takovém případě změna hodnoty formálního parametru způsobí změnu hodnoty skutečného parametru. Při předávání nevzniká kopie pole či objektu, ale kopie odkazu na pole či objekt.

V metodách mohou být deklarovány **lokální proměnné**. Princip deklarace je stejný, jako by se jednalo o běžnou proměnnou. Lokální proměnné vznikají při volání metody a zanikají po jejím ukončení. Na rozdíl od datových položek třídy není prováděna jejich automatická inicializace. Kompilátor však kontroluje, zda lokální proměnná byla inicializována. Hodnoty lokálních proměnných z předchozích volání metod se neuchovávají.

Rozsah platnosti metody definuje oblast programu, ve které je metoda přístupná (je ji možné volat). Jakmile vytvoříme v Javě metodu, je tato metoda

přístupná pouze ve třídě, v níž byla metoda definována. To znamená, že program nemůže volat metody z jiné třídy. Rozsah platnosti funkcí je možné měnit (určovat) pomocí modifikátorů (viz. kapitola 6):

- **public** - je viditelná všude, kde je viditelná třída, v které je metoda definovaná,
- **private** - je viditelná pouze ve své třídě,
- **protected** - je viditelná pouze ve své třídě a v jejích podtřídách a nebo v danném balíku.

NEZAPOMEŇTE

Více obecně, deklarace metody má v pořadí těchto 6 částí:

Modifikátory, jako jsou public, private a další.
Typ návratové hodnoty – datový typ hodnoty vrácené metodou nebo void, pokud metoda nevrací žádnou hodnotu.
Název metody – pravidla pro názvy atributů se dají aplikovat i na metody, ale konvence je mírně jiná.
Seznam parametrů v kulatých závorkách – čárkami oddělený seznam vstupních parametrů, předcházených jejimi typy, uzavřen mezi kulaté závorky. Jestliže metoda nepřebírá žádné parametry, musíte použít prázdné závorky.
Seznam výjimek – bude probírán později.
Tělo metody uzavřené mezi složené závorky – kód metody včetně deklarace lokálních proměnných patří sem.

Dvě z komponent deklarace metody tvoří **signaturu** (podpis) **metody** – název metody a typy parametrů.

5.2 METODA BEZ NÁVRATOVÉ HODNOTY

V praxi jsou tyto metody poměrně často používány. Bývají nazývány procedurami. Jejich návratový typ je void. Takové metody slouží např. pro realizaci vstupních či výstupních operací (tj. tiskové výstupy).

PŘÍKLAD 53

```
static void tisk ()
{
        System.out.println(Obvod je:+obv+ m);
}
```

5.3 METODY S VÍCE PARAMETRY

Metodě lze předat více než jeden parametr. Parametry navíc mohou být různého datového typu. Jejich počet by však neměl být příliš velký, taková metoda se stává nepřehlednou. Na první pohled do ní vstupuje takové množství heterogenních dat, že přestane být patrné, co metoda vykonává. Podívejme se na příklad provádějící výpočet odvěsny v pravoúhlém trohúhelníku za použití Pythagorovy věty.

PŘÍKLAD 54

```
static double pythagoras (double a , double b)
```

```
{
        return Math.sqrt(a*a+b*b);
}
// Metodu pythagoras() lze volat např. takto
double prepona=pythagoras(odvesna1, odvesna2);
```

5.4 IMPLICITNÍ A EXPLICITNÍ KONVERZE

Podívejme se podrobněji na problematiku implicitní a explicitní konverze mezi skutečnými a formálními parametry. Pokud je návratový typ metody odlišný od typu návratové hodnoty, je prováděna buď implicitní konverze (rozšiřující konverze) nebo explicitní konverze (zužující konverze). Implicitní konverze je prováděna kompilátorem automaticky, viz následující příklad.

PŘÍKLAD 55

```
static double pythagoras (int a , int b)
{
        double vysledek;
        vysledek=Math.sqrt(a*a+b*b);//rozsirujici konverze
        return vysledek;
}
```

Explicitní konverze musí být provedena uživatelem (možnost ztráty přesnosti dat), jinak nebude možno zdrojový kód zkompilovat.

PŘÍKLAD 56

```
static int pythagoras (double a , double b)
{
        double vysledek;
        vysledek=Math.sqrt(a*a+b*b);
        return (int) vysledek; //zuzujici konverze
}
```

5.5 METODY REKURZIVNÍ

Rekurze je schopnost metody volat sebe sama. Tento přístup se často uplatňuje při řešení některých typů problémů, např. z teorie her, zpracování dat, atd. Takovým typům problémů se říká **dekomponovatelné**. Můžeme je rozložit na podúlohy, realizující ty samé výpočty, ale pouze s jinými daty. Rekurzivně lze volat každou metodu s výjimkou metody main().

Rekurze nemůže probíhat do nekonečna. Musí existovat podmínka, za které je ukončena. Poté jsou zpětně dopočítávány hodnoty z předchozích volání metod. Typickým příkladem výpočtu za použití rekurze představuje faktoriál.

PŘÍKLAD 57

```
static int fakt(int n)
{
        if (n>1) return n*fakt(n-1);
        else return 1;
}
// Metodu lze volat např. takto:
        int cislo=10;
```

```
int faktorial=fakt(cislo);
```

5.6 POJMENOVÁNÍ METOD

Přestože jménem metody může být jakýkoli platný identifikátor, konvence omezují jména metody. Dle konvence může být názvem metody malými písmeny psané sloveso nebo víceslovný název, následovaný podstatnými jmény, přídavnými jmény atd. Ve víceslovných názvech by měla být první písmenka druhého a následujících slov velká. Zde je několik příkladů:

getBackground
compareTo
setX
bez
bezRychle
ziskejPozadi
ziskejKonecnaData
porovnejS
nastavX
jePrazdny

Typicky má každá metoda v rámci třídy své unikátní jméno. Nicméně v rámci třídy může mít více metod stejný název díky **přetížení metod**.

5.7 PŘETĚŽOVÁNÍ METOD

Přetěžování metod (více viz. kapitola 9.1)je jedním ze základních rysů OOP. Umožňuje deklarovat více metod stejného názvu, které se mohou lišit různým počtem, typem argumentů, popř. jejich pořadím. Postup se používá nejčastěji u metod provádějících stejné činnosti, ale s různými typy dat. Pokud zavoláme přetíženou metodu, kompilátor na základě typu parametrů, jejich pořadí či počtu vybere správnou metodu. Podívejme se na přetíženou metodu vzdalenost(), provádějící výpočet vzdálenosti mezi body zadané pravoúhlými souřadnicemi, souřadnicovými rozdíly popř. kombinací.

PŘÍKLAD 58

```
static double vzdalenost(double x1, double y1, double x2, double y2)
{
        return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
static double vzdalenost(double dx, double dy)
{
        return Math.sqrt(dx*dy+dy*dy);
}
static int vzdalenost(int x1, int y1, int x2, int y2)
{
        return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
}
```

Pokud budeme volat metodu vzdalenost() s následujícími parametry

double vzd=vzdalenost(1.0,0.0,6.0,0.0);

obdržíme výsledek 5.0. Bude provedena první metoda. Pokud budeme volat metodu vzdalenost() s následujícími parametry:

```
double vzd=vzdalenost(1,0,6,0);
```

obdržíme výsledek 5. Bude provedena třetí metoda.

Pokud budeme volat metodu vzdalenost() s následujícími parametry:

```
double vzd=vzdalenost(5.0,0.0);
```

bude provedena druhá metoda, obdržíme výsledek 5.0.

6 MODIFIKÁTORY PŘÍSTUPU KE ČLENŮM TŘÍD

Modifikátory přístupu slouží k tomu, abychom mohli pro danou třídu nastavit, které ostatní třídy mohou přistupovat ke členským proměnným a metodám dané třídy. Existují dvě úrovně kontroly přístupu:

- na úrovni tříd – modifikátor public (**veřejný**) nebo **přátelský modifikátor** (implicitně nastavený v případě, že neuvedeme jiný modifikátor)
- na úrovni členů tříd – modifikátor public, protected (**chráněný**) private (**soukromý**) nebo **přátelský modifikátor** (implicitně nastavený v případě, že neuvedeme jiný modifikátor)

Pokud třídu deklarujeme s modifikátorem public, potom je tato třída viditelná všude pro všechny třídy (tzn. můžeme ji použít v jakémkoli místě jakékoli jiné třídy). Pokud u třídy není uveden žádný specifikátor přístupu, tzn. je nastaven přátelský modifikátor, je tato třída viditelná pouze ve „svém“ balíčku (tzn. jen pro třídy, které se nacházejí ve stejném balíčku – z tohoto důvodu se někdy tento modifikátor taktéž nazývá jako **package-private**).

Na členské úrovni můžeme taktéž použít modifikátor public a přátelský modifikátor stejně jako na úrovni tříd a se stejným významem. Navíc však pro členy tříd existují ještě dva další modifikátory přístupu: private (**soukromý**) a protected (**chráněný**). Modifikátor private určuje, že daný člen je viditelný pouze ve své vlastní třídě. Modifikátor protected určuje, že daný člen je viditelný pouze ve svém balíčku (tzn. stejně jako *přátelský* modifikátor), ale navíc je ještě viditelný všemi podtřídami jeho třídy v ostatních balíčcích.

Následující tabulka ukazuje přístup ke členům třídy umožněný dle daného modifikátoru přístupu:

Tabulka 9: Úrovně přístupu

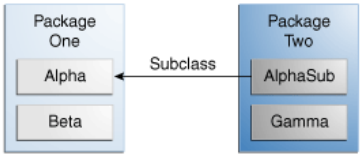
Modifikátor	Třída	Balíček	Podtřída	Svět
public	A	A	A	A
protected	A	A	A	N
přátelský	A	A	N	N
private	A	N	N	N

Ve sloupci Třída je uvedeno, kdy má samotná třída přístup k vlastnímu členu nastavenému s danou úrovní přístupu. Z tabulky lze vidět, že třída může vždy přistupovat k vlastnímu členu s libovolnou úrovní přístupu. Další sloupec uvádí, kdy je člen třídy přístupný (viditelný) pro třídy ze stejného balíčku (nezávisle na jejich nadřazenosti) pro danou úroveň přístupu. Další sloupec uvádí, kdy je člen třídy přístupný pro podtřídy dané třídy pro danou úroveň přístupu. Poslední sloupec uvádí, kdy je člen třídy přístupný pro všechny ostatní třídy pro danou úroveň přístupu.

Úrovně přístupu budeme při programování zvažovat hned dvakrát. Zaprvé, když budeme používat různé převzaté (námi nenapsané) Java třídy, úrovně přístupu nastavené pro tyto třídy a pro jejich členy nám budou určovat, které z nich můžeme ve svých vlastních třídách využívat. Zadruhé, při psaní vlastních tříd musíme u každé třídy a u každého jejího člena vždy zvážit, jakou úroveň přístupu mu nastavíme.

Podívejme se na soubor čtyř tříd a na to, jak úroveň přístupu ovlivňuje jejich vzájemnou viditelnost. Následující obrázek ukazuje, v jakém vztahu jsou tyto čtyři třídy:

Tabulka 10: Vzájemný vztah ukázkových čtyř tříd



Následující tabulka ukazuje, jak jsou členové třídy Alpha viditelní pro všechny uvedené třídy:

Tabulka 11: Viditelnost

Modifikátor	Alpha	Beta	Alphasub	Gamma
public	A	A	A	A
protected	A	A	A	N
přátelský	A	A	N	N
private	A	N	N	N

Nyní ještě uvedeme pár tipů pro volbu správné úrovně přístupu. Pokud ostatní programátoři používají naši třídu, chceme vždy zajistit, aby nemohla vzniknout chyba následkem jejího špatného použití. V tomto nám pomáhá nastavení úrovní přístupů.

- Používejme vždy tu nejvíce omezující úroveň přístupu, která má smysl pro daného člena třídy.
- Používejme vždy úroveň přístupu private, dokud nemáme důvod použít jinou.

Vyhýbejme se úrovním public až na konstanty. (Mnoho z příkladů uvedených v těchto skriptech používá úroveň public. Je to však z toho důvodu, abychom si některé věci vysvětlili na příkladech v co možná nejjednodušší formě. V praxi se však doporučuje používat modifikátor public velmi šetrně). Členové s úrovní public totiž často svádějí k nevhodnému propojování různých tříd, což potom velmi omezuje flexibilitu napsaného kódu při jeho budoucích úpravách.

7 KONSTRUKTORY, DESTRUKTORY

Konstruktor představuje speciální metodu používanou při inicializaci datových položek. V předcházejících kapitolách jsme použili poněkud nepohodlný způsob psaní zdrojového kódu ve tvaru objekt.polozka. Tuto konstrukci by v některých případech nebylo možno použít. K datovým položkám deklarovaných jako privátní bychom se nedostali z vnějšku třídy.

Při inicializování datových položek je proto používána specializovaná metoda zvaná konstruktor. Je volán v okamžiku vytvoření objektu.

PŘÍKLAD 59

```
Souradnice sour;
sour=new Souradnice(); //ted je volan konstruktor
```

Konstruktor má následující vlastnosti:

Jméno konstruktoru je shodné se jménem třídy.
Konstruktor **nemá žádnou návratovou hodnotu**, ani void. Nesmí obsahovat klíčové slovo return.
Konstruktor může mít **parametry** (explicitní konstruktor) nebo nemusí (implicitní konstruktor).
Konstruktor může být **přetížen**.

Konstruktor tedy umožňuje provádět inicializaci objektu při jeho vytvoření**. Implicitní konstruktor** je konstruktor bezparametrický. Pokud nevytvoříme konstruktor vlastní, je automaticky generován právě implicitní konstruktor. Ten umožňuje provést inicializaci datových položek na výchozí hodnoty. V našem případě by byly proměnné delka a uhel inicializovány na hodnoty 0.

Pokud bychom chtěli v rámci implicitního konstruktoru inicializovat proměnné na jiné než výchozí hodnoty, museli bychom si konstruktor napsat sami. Následující konstruktor provede inicializaci datových položek třídy na hodnoty 10 (délka) a 20 (úhel).

PŘÍKLAD 60

```
Souradnice()
{
        delka=10;
        uhel=20;
}
```

Pokud budeme vytvářet nové instance třídy Souradnice podle implicitního konstruktoru, všechny datové položky objektů budou inicializovány na stejné hodnoty. Podobného výsledku bychom mohli dosáhnout, pokud bychom upravili deklaraci třídy, a obě proměnné inicializovali zde.

PŘÍKLAD 61

```
public class Souradnice
{
        private double delka=10;
        private double uhel=20;
...
}
```

7.1 NÁZVY PARAMETRŮ

Když deklarujeme parametr metody nebo konstrukotru, musíme zadat **název parametru**. Toto jméno je používano uvnitř těla metody pro přístup k hodnotě příslušného argumentu. Název parametru musí být unikátní v jeho jmenném prostoru. Nemůže být stejný jako název jiného parametru pro stejnou metodu nebo konstruktor. Nemůže se jmenovat stejně jako lokální proměnná, deklarovaná v těle metody nebo konstruktoru.

Parametr může mít stejné jméno jako jeden z atributů. V tomto případě se říká, že parametr zakrývá atribut. Zakrývání atributů činí náš kód méně čitelným, a proto se dle konvenvce doporučuje používat pouze v konstruktoru nebo metodě pro nastavení odpovídajících atributů. Pro příklad si prohlédněme následující třídu Kruh a její metodu nastavitStred:

PŘÍKLAD 62

```
public class Kruh {
   private int x, y, radius;
   public void nastavitStred(int x, int y) {
      ...
   }
}
```

7.2 ODKAZ THIS

Existuje speciální typ odkazu, který má každá z metod. Odkaz this ukazuje na objekt, se kterým byla metoda volána. Tento odkaz je skrytý, není potřeba ho někde inicializovat. Zaručuje, že každá metoda pracuje s daty svého objektu. S odkazem this pracujeme také prostřednictvím tečkové notace. Pro bližší pochopení lze výše uvedený implicitní konstruktor zapsat prostřednictvím this.

PŘÍKLAD 63

```
Souradnice()
{
        this.delka=10;
        this.uhel=20;
}
```

7.3 EXPLICITNÍ KONSTRUKTOR

Explicitní konstruktor je konstruktor parametrický, disponuje formálními parametry. Umožňuje datovým složkám při inicializaci přiřadit libovolnou hodnotu. Explicitní konstruktor je volán s hodnotami parametrů, které budou přiřazeny formálním parametrům. Při inicializaci za použití explicitního konstruktoru může být

každý z objektů inicializován jinak, což při použití implicitního konstruktoru není možné. Podívejme na zápis explicitního konstruktoru.

PŘÍKLAD 64

```
Souradnice (double d, double u)
{
        delka=d;
        uhel=u;
}
```

Chceme-li vytvořit nový objekt a inicializovat ho explicitním konstruktorem, můžeme postupovat např. takto:

PŘÍKLAD 65

```
Souradnice sour;
sour=new Souradnice(10, 20); //volan explicitni konstruktor
```

S využitím odkazu this lze explicitní konstruktor přepsat do tvaru, ve kterém mají proměnné instance stejné jméno jako formální parametry.

PŘÍKLAD 66

```
Souradnice (double delka, double uhel)
{
        this.delka=delka;
        this.uhel=uhel;
}
```

Odkaz this tedy umožní rozlišit obě proměnné tak, že nedojde ke konfliktu jejich identifikátorů. Na případech implicitního a explicitního konstruktoru je patrné, že konstruktory mohou být přetěžovány. Přetěžování explicitních konstruktorů se řídí stejnými zákonitostmi jako přetěžování ostatních metod. Nezapomeňme, že i konstruktor je pouze metodou.

Odkaz this je možno použít i při inicializaci jednoho objektu objektem jiným za použití explicitního konstruktoru. Na rozdíl od C++ není nutno v takovém případě vytvářet kopírovací konstruktor. Podívejme se na následující příklad, ve kterém definujeme explicitní konstruktor ve tvaru:

PŘÍKLAD 67

```
Souradnice (Souradnice s)
{
        this.delka=s.delka;
        this.uhel=s.uhel;
}
```

Takový explicitní konstruktor má jako formální parametr odkaz na jiný objekt třídy. Chceme-li vytvořit objekt tak, aby byl při svém vzniku inicializován jiným objektem, budeme odkaz na tento objekt předávat jako parametr explicitnímu konstruktoru.

PŘÍKLAD 68

```
Souradnice sour=new Souradnice(10, 20);
```

```
//explicitni inicializace objektu sour2 objektem sour
Souradnice sour2=new Souradnice(sour);
```

Tento postup je v OOP velmi často používán, s jeho pomocí můžeme vytvářet poměrně snadno kopie objektů. Čtenář by se s ním měl proto nejen seznámit, ale měl by se ho naučit i používat.

V těle konstruktoru lze volat i metody. Tento přístup se používá v případě, kdy je nutno provést složitější inicializaci datových položek (např. tak, že hodnoty jsou jim nastaveny nějakým funkčním předpisem). Podívejme se na následující, byť poněkud umělý případ.

PŘÍKLAD 69

```
void NastavParametry(double d, double u)
{
        this.delka=d;
        this.uhel=u;
}
Souradnice (double d, double u)
{
        NastavParametry (d,u);
}
// Objekt následně inicializujeme stejným způsobem
Souradnice sour=new Souradnice(10, 20);
```

7.4 GETTERY A SETTERY

Dodržujeme -li princip zapouzdření, k datovým položkám třídy (deklarovaným jako private) můžeme přistupovat a manipulovat s nimi pouze prostřednictvím metod (deklarovaných jako public). V Javě jsou standardizovány názvy pro určité typy metod.

Pro nastavení hodnot datových položek jsou používány tzv. **settery**, tj. metody mající ve svém názvu slovo set. V případě vzorové třídy můžeme pro nastavení hodnot datových položek delka a uhel vytvořit metody:

PŘÍKLAD 70

```
public void setDelka(double d)
{
        this.delka=d;
}
public void setUhel(double u)
{
        this.uhel=u;
}
```

Pro získávání hodnot datových položek bývají používány tzv. **gettery**, tj. metody mající ve svém názvu slovo get. V případě vzorové třídy můžeme pro získání hodnot datových položek delka a uhel vytvořit metody:

PŘÍKLAD 71

```
public double getDelka() {return this.delka;}
public double getUhel() {return this.uhel;}
```

Nic nám nebrání pojmenovat metody pro nastavení a získání hodnot jinak, nicméně výše uvedený způsob je v Javě velmi často používán. Navíc řada vývojových prostředí umí automaticky generovat pro datové položky gettery a settery.

7.5 POLE OBJEKTŮ

Pracujeme-li s větším množstvím instancí jedné třídy, je vhodné ukládat je do **pole**. Práce s polem objektů je podobná jako práce s polem primitivních datových typů. Nestačí však vytvořit pouze pole, musíme provést i druhý krok, a tím je vytvoření instancí jednotlivých objektů prostřednictvím operátoru new.

K jednotlivým objektům lze následně přistupovat prostřednictvím **indexu**. Postup budeme ilustrovat na praktickém případě třídy Souradnice. Budeme vytvářet pole 10 objektů. V prvním kroku deklarujeme **odkaz** ukazující na pole objektů.

Souradnice [] pole;

Následně vytvoříme pole pole odkazů na objekty, kterým inicializujeme odkaz:

pole=new Souradnice[10];

V tomto okamžiku však jednotlivé prvky pole neukazují na žádné konkrétní objekty, jejich hodnota je null. Teprve nyní vytvoříme jednotlivé objekty, odkazy na ně inicializujeme položky pole. Inicializace proměnných instance může proběhnout jak prostřednictvím implicitního, tak prostřednictvím explicitního konstruktoru. V našem případě proběhne inicializace prostřednictvím explicitního konstruktoru.

PŘÍKLAD 72

```
for (int i=0;i<pole.length;i++)
{
        pole[i]=new Souradnice(10,20);
}
```

S jednotlivými objekty můžeme pracovat prostřednictvím odkazů, Jednotlivé hodnoty souřadnic x a y můžeme např. vypsat za použití cyklu.

PŘÍKLAD 73

```
for (int i=0;i<pole.length;i++)
{
        System.out.println(pole[i].x()," ", pole[i].y());
        pole[i]=new Souradnice(10,20);
}
```

7.6 PŘEDÁVÁNÍ OBJEKTŮ JAKO PARAMETRŮ

Objekty jsou na rozdíl od primitivních datových typů v Javě předávány odkazem. Pro připomenutí uveďme, že se tento přístup týká i práce s poli. Formálním parametrem v metodě je odkaz na objekt třídy, tj. **reference**. Skutečný parametr představuje odkaz na již existující objekt. Oba odkazy by měly být stejného datového typu. Prostřednictvím reference může metoda přistupovat k metodám volajícího

objektu (popř. k datovým položkám, nejsou–li deklarovány jako privátní). Tímto způsobem můžeme prostřednictvím formálního parametru měnit hodnoty skutečného parametru, tj. volající data.

Tento přístup ilustrujeme na nové třídě Prevod, kterou vytvoříme. Bude v ní implementována metoda vzdalenost(), která zpětně z pravoúhlých souřadnic spočte jednu z polárních souřadnic: vzdálenost.

PŘÍKLAD 74

```
public class Prevod
{
public double vzdalenost (Souradnice objekt)
{
return Math.sqrt(Math.power(objekt.x(),2)+Math.power(objekt.y(),2));
}
}
```

Metoda má jako formální parametr odkaz na objekt třídy Souradnice. Prostřednictvím formálního parametru přistupuje k metodám x() a y() volajícího objektu. V metodě main() třídy Souradnice vytvoříme dva objekty.

PŘÍKLAD 75

```
public static void main (String [] args)
{
//Vytvoreni instanci
Souradnice sour=new Souradnice(10, 20);
Prevod prev=new Prevod();
//Volani metody a predani odkazu na objekt
System.out.println(Vzdalenost=+prev.vzdalenost(sour));
}
```

Metodě vzdalenost() předáme odkaz na objekt sour, spočtenou vzdálenost následně vytiskneme.

7.7 DESTRUKCE OBJEKTU

Opakem konstrukce objektu je jeho **destrukce**. V některých jazycích si musíme o zničení objektu a uvolnění paměti, kterou zabírá, explicitně zažádat. V Javě existuje **garbage collection**, což je automatická fáze v běhu programu, během které se zničí všechny objekty, na něž nemůžeme syntakticky nijak dosáhnout.

O garbage collection můžeme sice také zažádat pomocí volání System.gc(), ale není zaručeno, že nám virtuální stroj vyhoví a skutečně kolekci zavolá, případně že zničí i náš objekt (garbage collection je velmi složitý proces). Z tohoto důvodu se volání kolekce vesměs nechává na virtuálním stroji, aby se sám rozhodl, kdy je nejvhodnější okamžik.

Podobně jako existuje konstruktor, tak v Javě existuje metoda finalize(), kterou mají všechny objekty, a jež je volána jako poslední před zničením objektu. Tuto metodu můžeme ve svých objektech překrýt (znovu deklarovat a implementovat), ale její použití je vhodné pouze pro velmi specifický kód, jelikož nevíme, kdy přesně dojde k jejímu zavolání.

8 DĚDIČNOST

Dědičnost je jeden z hlavních principů objektově orientovaného programování. Jeho základní myšlenka spočívá v následujícím principu – každá třída má svého předka, po kterém dědí veškeré proměnné a metody.

V Javě má každá třída právě jednoho předka (na rozdíl např. od C++). Každá třída obsahuje všechny veřejné proměnné i funkce, které obsahovala třída předka. K nim může dodefinovat nové proměnné a funkce. Dědičností jsou třídy uspořádány do stromové struktury, kde na vrcholu stromu je třída Object (ta je v Javě jedinou třídou, která nemá předchůdce a všechny třídy jsou jejími přímými nebo nepřímými následníky).

Dědičnost se deklaruje pomocí klíčového slova extends uvedeného za jménem třídy. Pokud není uvedeno, je deklarovaná třída následníkem třídy Object.

8.1 UKÁZKA DEDIČNOSTI V KÓDU

Budeme programovat informační systém. To je docela reálný příklad, abychom si však učení zpříjemnili, bude to informační systém pro správu zvířat v ZOO. Náš systém budou používat dva typy uživatelů: uživatel a administrátor. Uživatel je běžný ošetřovatel zvířat, který bude moci upravovat informace o zvířatech, např. jejich váhu nebo rozpětí křídel. Administrátor bude moci také upravovat údaje o zvířatech a navíc zvířata přidávat a mazat z databáze. Z atributů bude mít navíc telefonní číslo, aby ho bylo možné kontaktovat v případě výpadku systému. Bylo by jistě zbytečné a nepřehledné, kdybychom si museli definovat obě třídy úplně celé, protože mnoho vlastností těchto dvou objektů je společných. Uživatel i administrátor budou mít jistě jméno, věk a budou se moci přihlásit a odhlásit. Nadefinujeme si tedy pouze třídu Uzivatel.

PŘÍKLAD 76

```
class Uzivatel
{
	private String jmeno;
	private String heslo;
	private int vek;

	public boolean prihlasit(String heslo)
	{
		...
	}

	public boolean odhlasit
	{
		...
	}

	public void nastavVahu(Zvire zvire)
	{
		...
	}

	...
}
```

Třídu jsme jen naznačili, ale jistě si ji dokážeme dobře představit. Bez znalosti dědičnosti bychom třídu Administrator definovali asi takto:

PŘÍKLAD 77

```
class Administrator
{
	private String jmeno;
	private String heslo;
	private int vek;
	private String telefonniCislo;

	public boolean prihlasit(String heslo)
	{
		...
	}

	public boolean odhlasit
	{
		...
	}

	public void nastavVahu(Zvire zvire)
	{
		...
	}

	public void pridejZvire(Zvire zvire)
	{

	}

	public void vymazZvire(Zvire zvire)
	{

	}

	...
}
```

Vidíme, že máme ve třídě spoustu redundantního (duplikovaného) kódu. Jakékoli změny musíme nyní provádět v obou třídách, kód se nám velmi komplikuje. Nyní použijeme dědičnost, definujeme tedy třídu Administrator tak, aby z třídy Uzivatel dědila. Atributy a metody uživatele tedy již nemusíme znovu definovat, Java nám je do třídy sama dodá:

PŘÍKLAD 78

```
class Administrator extends Uzivatel
{
        private String telefonniCislo;

        public void pridejZvire(Zvire zvire)
        {

        }

        public void vymazZvire(Zvire zvire)
        {

        }

        ...
}
```

Vidíme, že ke zdědění jsme použili klíčové slovo extends. V anglické literatuře najdete dědičnosti pod slovem **inheritance**.

V příkladu výše se nepodědí soukromé atributy, ale pouze atributy a metody s modifikátorem public. Soukromé atributy a metody jsou chápány jako speciální logika konkrétní třídy, která se nedědí. Abychom dosáhli požadovaného výsledku, použijeme nový modifikátor přístupu protected, který funguje stejně, jako private, ale dovoluje tyto atributy dědit. Začátek třídy Uzivatel by tedy vypadal takto:

PŘÍKLAD 79

```
class Uzivatel
{
        protected String jmeno;
        protected String heslo;
        protected int vek;

        ...
```

Když si nyní vytvoříme instance uživatele a administrátora, oba budou mít např. atribut jmeno a metodu prihlasit(). Třída Uzivatel zdědí všechny atributy třídy Administrator.

Výhody dědění jsou jasné, nemusíme opisovat pro obě třídy ty samé atributy, ale stačí dopsat jen to, v čem se liší. Zbytek se podědí. Přínos je obrovský, můžeme rozšiřovat existující komponenty o nové metody a tím je znovu využívat. Nemusíme psát spousty redundantního (duplikovaného) kódu. A hlavně, když změníme jediný atribut v mateřské třídě, automaticky se tato změna všude podědí. Nedojde tedy k tomu, že bychom to museli měnit ručně u 20ti tříd a někde na to zapomněli a způsobili chybu. Jsme lidé a chybovat budeme vždy, musíme tedy používat takové programátorské postupy, abychom měli možností chybovat co nejméně.

O mateřské třídě se někdy hovoří jako o **předkovi** (zde Uzivatel) a o třídě, která z ní dědí, jako o **potomkovi** (zde Administrator). Potomek může přidávat nové metody nebo si uzpůsobovat metody z mateřské třídy (viz dále). Můžeme se setkat i s pojmy **nadtřída** a **podtřída**.

Další možností, jak objektový model navrhnout, by bylo zavést mateřskou třídu Uzivatel, která by sloužila pouze k dědění. Z Uzivatel by potom dědily Osetrovatel a z něj Administrator. To by se však vyplatilo při větším počtu typů uživatelů. V takovém

případě hovoříme o **hierarchii tříd**. Náš příklad byl jednoduchý a proto nám stačily pouze 2 třídy. Existují tzv. **návrhové vzory**, které obsahují osvědčená schémata objektových struktur pro známé **případy užití**. V objektovém modelování se dědičnost znázorňuje graficky jako **prázdná šipka směřující k předkovi**.

8.2 Datový typ při dědičnosti

Obrovskou výhodou dědičnosti je, že když si vytvoříme proměnnou s datovým typem mateřské třídy, můžeme do ní bez problému ukládat i její potomky. Je to dané tím, že potomek obsahuje vše, co obsahuje mateřská třída, splňuje tedy "požadavky" (přesněji obsahuje rozhraní) datového typu. A k tomu má oproti mateřské třídě něco navíc. Můžeme si tedy udělat pole typu Uzivatel a v něm mít jak uživatele, tak administrátory. S proměnnou to tedy funguje takto:

PŘÍKLAD 80

```
Uzivatel u = new Uzivatel("Jan Novák", 33);
Administrator a = new Administrator("Josef Nový", 25);
// Nyní do uživatele uložíme administrátora:
u = a;
// Vše je v pořádku, protože uživatel je předek
// Zkusíme to opačně a dostaneme chybu:
a = u;
```

V Javě je mnoho konstrukcí, jak operovat s typy instancí při dědičnosti. Nyní si ukažme jen to, jak můžeme ověřit typ instance v proměnné:

PŘÍKLAD 81

```
uzivatel u = new Administrator("Josef Nový", 25);
if (u instanceof Administrator)
        System.out.println("Je to administrátor");
else
        System.out.println("Je to uživatel");
```

Pomocí operátoru instanceof se můžeme zeptat, zda je objekt daného typu. Kód výše otestuje, zda je v proměnné u uživatel nebo jeho potomek administrátor.

Jazyky, které dědičnost podporují, buď umí dědičnost jednoduchou, kde třída dědí jen z jedné třídy, nebo vícenásobnou, kde třída dědí hned z několika tříd najednou. Vícenásobná dědičnost se v praxi příliš neosvědčila. Java podporuje pouze jednoduchou dědičnost, s vícenásobnou dědičností se můžeme setkat např. v C++.

8.3 Dědění s konstruktorem, super

Dědení je jednoduché v případě, že by nebyl v předkovi konstruktor. Stejná situace by byla i v případě, že by obsahoval konstruktor bez parametrů (zdědil by ho z třídy Object). Pokud by předek obsahoval konstruktor s parametry, bylo by v potomkovi nutno napsat také konstruktor, a jeho první příkaz musí být volání konstruktoru předka (supertřídy).

Mějme předka s konstruktorem s parametry public Plocha(int stranaA):

PŘÍKLAD 82

```
public class Plocha {
	private int stranaA;

	public Plocha(int stranaA) {
		this.stranaA = stranaA;
	}
	public int plocha() {
		return stranaA*stranaA;
	}
}
```

Má-li předek konstruktor s nejméně jedním parametrem, musíme v potomkovi taky napsat konstruktor, jeho první příkaz bude super():

PŘÍKLAD 83

```
public class Objem extends Plocha {
	private int vyskaH;

	public Objem(int vyskaH, int stranaA) {
		super(stranaA);
		this.vyskaH = vyskaH;
	}

	public int objem() {
		return vyskaH*plocha();
	}
}
```

Konstruktor každé třídy musí na začátku obsahovat volání konstruktoru třídy předka, aby se zajistila řádná inicializace zděděných proměnných (tzv. řetězení konstruktorů). K volání konstruktoru předka slouží klíčové slovo super.

super(parametryKonstruktoru);

Toto volání musí být provedeno jako první, aby se zajistila řádná inicializace v třídě předka. Volání konstruktoru předka lze vynechat pouze v případě, že třída předka má pouze jeden konstruktor bez parametrů nebo konstruktor implicitní. Pak toto volání překladač na začátek konstruktoru sám doplní.

Při jednotlivých voláních metod podtypů často nazazíme na to, že nechceme celou metodu překrýt, pouze k ní chceme přidat další funkcionalitu. V tento okamžik můžeme zavolat super.jmenoMetody(), čímž zavoláme funkcionalitu předka. Analogicky k řetězení konstruktorů this() můžeme volat konstruktor předka voláním super() – toto volání musí být v rámci konstruktoru potomka vždy na prvním místě.

Dalším aspektem volání kostruktoru v rámci hierarchie je, že není příliš vhodné volat kteroukoliv metodu, která může být překryta. Při volání konstruktoru se objekt vytváří postupně z vrchu hierarchie (tzn. od třídy Object). Pokud bychom proto volali metodu, která je překrytá v některém z potomků, a která zde využívá pole, které ještě nemohly být inicializovány, tak bychom se mohli dočkat havárie programu.

8.4 TŘÍDA OBJECT

Třída Object z balíku java.lang je kořenovou třídou ve stromu tříd, tj. všechny třídy, ať už knihovní nebo navržené programátorem, mají společného (nepřímého) předka třídu Object.

Object definuje základní metody, které musí mít každý objekt v Javě - většinu z nich používá runtime systém. Jedná se o metody:

- protected native clone() - vytvoří identický objekt (ale nevolá konstruktor) a přiřadí stejné hodnoty všem členským proměnným. Funguje pouze u tříd, které implementují rozhraní Cloneable.
- public boolean equals(Object obj) - porovnává objekt s objektem obj (způsob porovnávání se pro jednotlivé potomky liší).

- public final Class getClass() - vrací objekt reprezentující třídu instance v runtime systému.
- protected void finalize() throws Throwable - je volána při úklidu; standardně neprovádí nic.
- public int hashCode() - používá se k hašování, není-li překrytá vrací totéž co metoda System.identityHashCode(). Metoda hashCode() vrací pro každou instanci číslo typu int. Využívá se pro optimalizaci ukládání do dynamických datových struktur HashSet nebo HashTable.
- public String toString() - vrací identifikační řetězec objektu (často předefinovávaná metoda pro účely ladění).
- wait(), notify(), notifyAll() - tyto metody se vztahují k použití více vláken (thread) v programu. Problematika vláken je mimo rozsah těchto skript.

9 POLYMORFISMUS

Pojem **polymorfismus** pochází z řečtiny a znamená mnohotvarost. V objektovém programování vyjadřuje situaci, kdy se při stejném volání provádí různý kód. Která konkrétní metoda se provede, závisí na:

- předávaných parametrech,
- objektu, kterému je předávána.

První varianta polymorfismu je v Javě realizována přes tzv. **přetěžování metod**. Druhá varianta polymorfismu, která je v Javě závislá na dědičnosti a **pozdní vazbě**[3], se realizuje přes tzv. **překrývání metod**.

9.1 PŘETĚŽOVÁNÍ METOD

Programovací jazyk Java podporuje přetížené metody a může metody od sebe rozeznat pomocí odlišných signatur. To znamená, že **v jedné třídě mohou existovat metody stejného jména jen v případě, že každá má jiný seznam parametrů**.

Představme si, že máme třídu, která pomocí kaligrafie[4] vykresluje různé typy dat (řetězce, celá čísla atd.) a obsahuje metody pro vykreslení každého datového typu. Je nešikovné použít pro každou metodu nový název – například, nakresliString, nakresliInteger, nakresliFloat atd. V programovacím jazyce Java, můžeme použít pro kreslicí metody stejné názvy, ale musí se lišit jejich seznam parametrů. Tudíž ona kreslící třída bude definovat čtyři metody s názvem nakresli, každá s jiným seznamem parametrů.

PŘÍKLAD 84

```
public class KreslicDat {
        ...
        public void nakresli(String s) {
                ...
        }
        public void nakresli(int i) {
                ...
        }
        public void nakresli(double f) {
                ...
        }
        public void nakresli(int i, double f) {
                ...
        }
}
```

Přetížené metody jsou odlišené podle počtu a typu parametrů, předaných metodě. Ve výše uvedeném příkladě, nakresli(String s) a nakresli(int i) jsou rozeznány jako unikátní metody, protože mají rozdílný typ parametru. Nemůžeme deklarovat více metod se stejným názvem a stejným počtem a typem parametrů,

[3] V případě pozdní vazby je typ objektu, na němž bude metoda volána, rozhodnut až za běhu programu, nikoliv během jeho kompilace.
[4] umění krasopisu

protože kompilátor by je nemohl od sebe odlišit. Kompilátor nerozlišuje metody podle typu jejich návratové hodnoty, takže nemůžeme deklarovat metody se stejnou signaturou, ačkoli by se lišily v typu návratové hodnoty. Nakonec jedna poznámka: přetížené metody by se měly používat šetrně, protože jejich použití činí kód obtížně čitelný.

9.2 PŘEKRÝVÁNÍ METOD

Jak již bylo napsáno na začátku této kapitoly, překrývání metod souvisí s dědičností tříd, kterou jsme si popsali v již předchozí kapitole. Tam jsme si uvedli její jednu z hlavních výhod: tím, že potomek nějaké třídy může využívat všechny metody předka, nemusíme metody, které jsou shodné jak u předka, tak u potomka, psát vícekrát, čímž můžeme náš kód velice zjednodušit. Potomek však může zdědit metody, které nám nemusí vyhovovat, a chtěli bychom je nahradit jinými. Právě pro tento účel slouží překrývání metod.

Metody se překrývají tak, že v potomkovi napíšeme jiné tělo metody zděděné od svého předka, tzn. opíšeme pouze hlavičku metody, tedy název, počet parametrů a jejich typy, a tělo již napíšeme dle svého uvážení. Při volání překryté metody u potomka se poté metoda se shodnou hlavičkou u předka ignoruje a zavolá se překrytá metoda.

Vysvětleme si to na příkladu. Definujme si třídu Predek.

PŘÍKLAD 85

```
public class Predek {
   protected double stranaA;

  public Predek(double stranaA) {
    this.stranaA = stranaA;
  }

  public double plocha() {
    return stranaA*stranaA;
  }
}
```

Dále si definujme třídu Potomek – ta bude také obsahovat metodu plocha(),

PŘÍKLAD 86

```
public class Potomek extends Predek {
  private double stranaB;

  public Potomek(double stranaB, double stranaA) {
    super(stranaA);
    this.stranaB = stranaB;
  }

  public double plocha() {
    return stranaB*stranaA;
  }
}
```

která překryje metodu předka. Při volání metody bude použita metoda z potomka. Metodu předka tedy již k dispozici pomocí standardního volání nemáme. Někdy se však může stát, že ji budeme potřebovat – např. metoda potomka vychází z výsledků metody předka a dále ho již jen nějak upravuje. V takovém případě můžeme využít klíčové slovo super, které nám umožňuje přistupovat k překrytým (k nepřekrytým můžeme přistupovat přímo v potomkovi) metodám předka (viz. Kapitola 8.3).

9.3 ROZHRANÍ

9.3.1 CO JE TO ROZHRANÍ

Rozhraní (**interface**) je množina metod, která může být implementována třídou. Interface pouze popisuje metody, jejich vlastní implementace však neobsahuje.

V Javě na rozdíl od jiných programovacích jazyků (například C++) nemůže třída dědit od více tříd najednou (neexistuje vícenásobná dědičnost). Každá třída však může implementovat libovolný počet rozhraní, do jisté míry tedy rozhraní vícenásobnou dědičnost nahrazují. Implementace rozhraní není na hierarchii tříd nijak vázána a nevzniká z ní vztah dědičnosti, o kterém jsme mluvili v minulé kapitole.

9.3.2 DEFINICE ROZHRANÍ

Definice rozhraní je vidět na následujícím příkladu. Hlavička se skládá z modifikátoru viditelnosti (public), klíčového slova interface a jména rozhraní. Vlastní tělo definice pak obsahuje definici metod rozhraní.

PŘÍKLAD 87

```
import java.awt.*;
public interface Visualni {
  public void vykresli (Graphics kam);
}
```

Rozhraní může kromě definic metod obsahovat také konstanty. Ty se pak chovají stejně, jako by se jednalo o konstanty třídy, která toto rozhraní implementovala.

Podobně jako lze pomocí dědičnosti rozšiřovat třídy, lze rozšiřovat i rozhraní. Pokud vytvoříme rozhraní, které dědí od jiného rozhraní, automaticky tak přebírá všechny jeho metody a konstanty. Dědičnost rozhraní se zapisuje klíčovým slovem extends. Rozhraní také mohou dědit od více rozhraní najednou.

PŘÍKLAD 88

```
public interface Posuvne {
  public void posun (float dx, float dy);
}
public interface Zvetsujici {
  public void zvetsi (float kolikrat);
}
public interface Pohyblive extends Zvetsujici, Posuvne { }
```

9.3.3 IMPLEMENTACE ROZHRANÍ

Implementovat rozhraní znamená implementovat všechny metody, které toto rozhraní definuje. Pokud třída implementuje nějaké rozhraní, je tím zaručeno, že obsahuje definici všech, které rozhraní definuje.

Implementace rozhraní se uvádí v záhlaví definice třídy klíčovým slovem implements.

PŘÍKLAD 89

```
import java.awt.*;
public class VisualniKruznice implements Visualni {
  public void vykresli (Graphics kam) {
    ...
    }
  ...
}
```

Třída může implementovat libovolný počet rozhraní, jak ukazuje následující příklad:

PŘÍKLAD 90

```
import java.awt.*;
public class VisualniPohyblivyCtverec implements Visualni, Pohyblive {
public void vykresli (Graphics kam) {
  ...
  }
public void posun (float dx, float dy) {
  ...
  }
public void zvetsi (float kolikrat) {
  ...
  }
}
```

9.3.4 POUŽITÍ ROZHRANÍ JAKO TYPŮ

V Javě lze definovat proměnnou typu reference na rozhraní, ve které může být uložena libovolná třída, která toto rozhraní implementuje. Jména rozhraní lze používat jako referenční datové typy stejným způsobem jako jména tříd.

```
Visualni visualniObjekt = new VisualniKruznice();
visualniObjekt = new VisualniCtverec();
```

9.3.5 PŘÍKLADY ROZHRANÍ

Pro psaní jednoduchých aplikací nebudeme zpravidla potřebovat definovat vlastní rozhraní, určitě se však neobejdeme bez použití (implementace) předdefinovaných rozhraní Javy. Tady je seznam některých standardních často používaných rozhraní jazyka Java:

- **java.awt.LayoutManager** - toto rozhraní definuje metody, které jsou nezbytné k tomu, aby třída mohla uspořádat grafické objekty na ploše okna. Blíže se s ním setkáme v díle, který bude věnovaný knihovně AWT.
- **java.io.DataInput**, **java.io.DataOutput** - tato rozhraní definují metody požadované, pokud objekt chce komunikovat pomocí proudů pro předávání primitivních datových typů nebo znakových dat.
- **java.lang.Runnable** - toto rozhraní definuje metodu run, jejíž kód běží jako samostatné vlákno

9.4 ABSTRAKTNÍ TŘÍDY

V některých situacích je výhodné vytvořit jedinou bázovou třídu pro více tříd odpovídajících konkrétním objektům, i když tato samotná bázová třída žádnému konkrétnímu objektu neodpovídá. Může ovšem nést některá data a poskytovat metody, které jsou odvozeným třídám společné. Taková třída se pak nazývá abstraktní a je označena klíčovým slovem abstract. Překladač jazyka Java pak zajistí, že instanci abstraktní třídy nelze operátorem new přímo vytvořit, mohou se vytvářet pouze instance konkrétních tříd.

Abstraktní třída může deklarovat některé společné metody a poskytovat jejich základní implementaci. Pokud odvozená třída takovou metodu nepředefinuje, pak se pro její instance použije implementace poskytnutá v bázové třídě.

Mohou však nastat i situace, kdy skutečně vyžadujeme, aby odvozené třídy určitou metodu vždy definovaly. Takovou metodu pak také nazýváme abstraktní a označujeme klíčovým slovem abstract, navíc u ní není uvedeno tělo a hlavička metody je zakončena středníkem. Pokud odvozená třída některou abstraktní metodu neimplementuje, musí být také označena jako abstraktní. Tím je zajištěno, že instance konkrétních tříd mají všechny metody implementované.

Uveďme si nyní využití abstraktních metod na jednoduchém příkladu, ve kterém vytvoříme třídu reprezentující obecný obrazec se souřadnicemi středu, obvodem a plochou a na jejím základě definujeme třídy pro kruh, obdélník a čtverec.

Obrazec je abstraktním pojmem, pro který nemá smysl definovat implicitní metodu pro výpočet obvodu a plochy. Definujeme jej tedy jako abstraktní třídu poskytující pouze souřadnice středu a abstraktní metody pro výpočet obvodu a plochy:

PŘÍKLAD 91

```
abstract class Obrazec {
  public Obrazec(double x, double y) {
    this.x = x; this.y = y;
  }
  public abstract double obvod();
  public abstract double obsah();

  protected double x;
  protected double y;
}
```

Měli bychom také doplnit metody pro získání a nastavení hodnot souřadnic středu, ale zatím se bez nich obejdeme.

Pro reprezentaci kruhu jen poněkud upravíme řešení, které jsme vytvořili dříve:

PŘÍKLAD 92

```
class Kruh extends Obrazec {

   public Kruh(double x, double y, double r) {
      super(x, y);
      this.r = r;
   }
   public double obvod() { return 2 * 3.14159 * r; }
   public double obsah() { return 3.14159 * r * r; }

   protected double r;
}
```

Třída reprezentující obdélník doplní k abstraktnímu obrazci údaje o délkách stran a implementuje výpočet obvodu a obsahu:

PŘÍKLAD 93

```
class Obdelnik extends Obrazec {
   public Obdelnik(double x, double y, double a, double b) {
      super(x, y);
      this.a = a; this.b = b;
   }
   public double obvod() { return 2 * (a + b); }
   public double obsah() { return a * b; }

   protected double a;
   protected double b;
}
```

Konečně třídu reprezentující čtverec již můžeme odvodit přímo z obdélníka vhodným voláním jeho konstruktoru:

PŘÍKLAD 94

```
class Ctverec extends Obdelnik {
   public Ctverec(double x, double y, double a) {
      super(x, y, a, a);
   }
}
```

Výsledné řešení můžeme otestovat takovým způsobem, že vytvoříme pole několika obrazců a pak pro ně postupně vypočteme jednotlivé hodnoty. Povšimněme si toho, jakým způsobem můžeme inicializovat pole objektů (v Javě je pole také objekt, proto se vytváří operátorem new stejně jako jiné objekty).

PŘÍKLAD 95

```
class TestObrazcu {
   public static void main(String[] args) {
      Obrazec[] pole = new Obrazec[] {
            new Obdelnik(0,0,2,5),
            new Ctverec(0,0,4),
            new Kruh(0,0,1) };
      for(int i = 0; i < pole.length; i++ ) {
         System.out.print  ("pole["+i+"]:");
         System.out.print  (" obvod=" + pole[i].obvod());
         System.out.println(" obsah=" + pole[i].obsah());
```

```
    }
  }
}
```

10 LADĚNÍ PROGRAMU

V předchozích částech textu jsme se seznámili se základy objektově orientovaného programování v jazyce Java. Vybaveni těmito znalostmi jsme již schopni vytvořit již značné množství objektově navržených programů řešících různé, i ne zcela triviální, úlohy. Jak však spolu s rostoucí složitostí řešené úlohy roste i složitost implementace programu řešícího tuto úlohu, je program stále více a více náchylnější pro vznik nějaké chyby, která poté způsobuje jeho nefunkčnost, či funkčnost nesprávnou.

V této kapitole se seznámíme s prostředky, které nám vzniklou chybu v programu mohou pomoci odhalit a následně nám umožní chybu opravit (tomuto procesu odhalování a opravování (nebo jen krátce **odstraňování**) vzniklých chyb se říká **ladění programu**, v angl. **debugging**).

Využíváme-li při psaní programů služeb nějakého vývojového prostředí, můžeme pro odstraňování chyb používat nástroje tohoto prostředí k tomu určené. Při výběru vývojového prostředí proto vždy dbejme na to, aby nám toto prostředí nabízelo co nejsilnější prostředky na odhalování a napravování vzniklých chyb.

V tomto výukovém materiálu jsme ve druhé kapitole popsali základy práce s vývojovým prostředí *Eclipse Indigo*, takže se nyní k tomuto prostředí v souvislosti s laděním programu opět vrátíme.

10.1 LADĚNÍ PROGRAMU V ECLIPSE INDIGO

Nejdříve si vytvoříme jednoduchý příklad, na kterém poté budeme ladění v Eclipse Indigo ilustrovat. Vytvořme třídy Counter

PŘÍKLAD 96

```
public class Counter {
  private int result = 0;

  public int getResult() {
    return result;
  }

  public void count() {
    for (int i = 0; i < 100; i++) {
      result += i + 1;
    }
  }
}
```

a Counting.

PŘÍKLAD 97

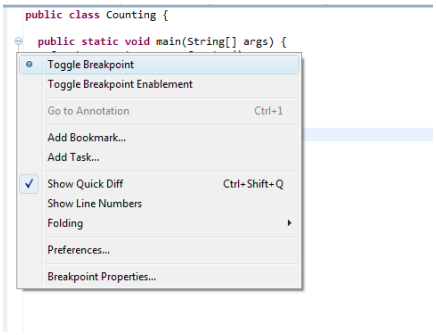
```
public class Counting {

  public static void main(String[] args) {
    Counter counter = new Counter();
    counter.count();
    System.out.println("We have counted "
        + counter.getResult());
  }
}
```

10.1.1 NASTAVENÍ ZARÁŽKY

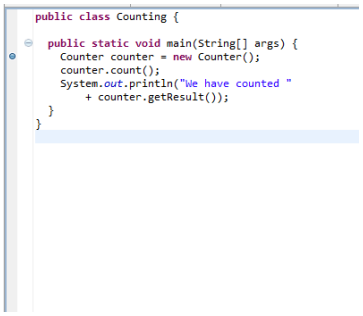
Abychom nastavili **zarážku** (angl. **breakpoint**), klikneme pravým tlačítkem myši na levý okraj okna se zdrojovým kódem a zvolíme *Toggle Breakpoint*. Popř. můžeme dvojkliknout v tom samém místě.

Obrázek 20: Nastavení zarážky



Jak lze vidět na následujícím obrázku, nastavili jsme zarážku na řádku Counter counter = new Counter();.

Obrázek 21: Zarážka nastavená na řádku Counter counter = new Counter();

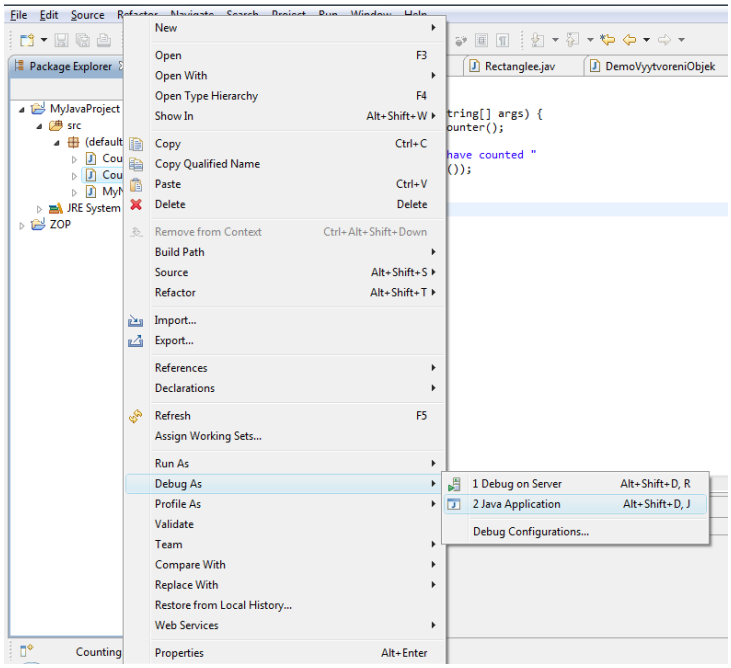


Zarážka nám umožňuje přerušit běh programu v určitém místě (např. v místě, kde by se mohla vyskytnout chyba). Jejich význam je popsán v následující podkapitole.

10.1.2 SPUŠTĚNÍ DEBUGGERU

Debugger je anglické označení nástroje pro odhalování chyb v programech, které se do českého jazyka nijak nepředkládá – budeme ho tedy používat v anglické formě. Abychom spustili debugger, nejdříve zvolíme Java soubor, který obsahuje metodu main(), potom klikneme pravým a zvolíme *Debug As → Java Application*.

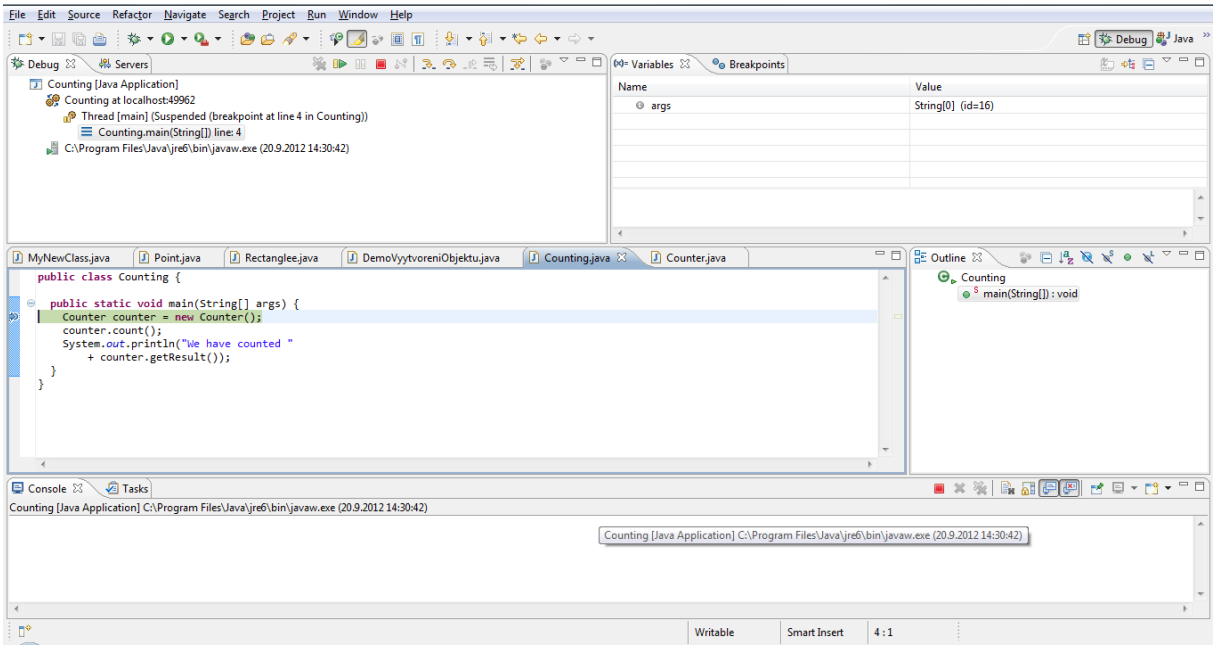
Obrázek 22: Spuštení debuggeru



Pokud bychom ve zdrojovém kódu neumístili žádné zarážky, program by proběhl jako „normálně“. K **debugování** (odstraňování chyb) je tedy potřeba umístit alespoň jednu zarážku.

Pokud spouštíme debugger poprvé, Eclipse se nás zeptá, zda chceme změnit perspektivu (rozložení a typ oken) na debugovací perspektivu. Odpovíme *Ano*. Poté bychom měli vidět perspektivu velmi podobnou té v následujícím obrázku.

Obrázek 23: Debug perspektiva



Program se spustí stejně jako při standardním spuštění přes volbu *Run*. Avšak v momentě, kdy zpracování programu „narazí“ na zarážku, zpracování se přerusí, avšak nezastaví – všechny proměnné, objekty, atp. zůstanou nastaveny na jejich

aktuální hodnotu, což nám umožní jejich kontrolu, zda jsou nastaveny dle našich očekávání. Pokud žádný rozpor oproti našim očekáváním nenalezneme, můžeme program nechat provést další část kódu, znovu překontrolovat stavy proměnných a tento postup opakovat do té doby, dokud nenarazíme na nějakou chybu nebo na konec programu – tomuto postupu se říká tzv. **krokování** programu, jelikož program kontrolujeme vždy po provedení určité části kódu (jednom kroku). Abychom si mohli nastavit velikost kroku – tedy kolik kódu má být provedeno před dalším přerušením – můžeme využít klávesy F5, F6, F7 a F8. Jejich význam je uveden v následující tabulce.

Tabulka 12: Význam kláves pro krokování kódu

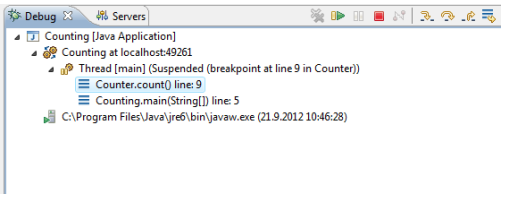
Klávesa	Popis
F5	Provede se příkaz na aktuálním (zeleném) řádku a poté se debugger na dalším řádku zastaví. Pokud aktuální řádek obsahoval funkci nebo metodu, debugger vstoupí do odpovídající funkce, reps. metody.
F6	Provede se příkaz na aktuálním (zeleném) řádku a poté se debugger na dalším řádku zastaví. Pokud aktuální řádek obsahoval funkci nebo metodu, tato se provede celá, aniž by v ní debugger zastavil.
F7	Provede se celý zbytek aktuální metody a debugger zastaví na následující části kódu v nadřazené metodě.
F8	Program se provádí, dokud nenarazí na další zarážku. Pokud na další zarážku nenarazí, probíhá normálně dál.

Krokování můžeme také provádět skrze ikony v horní části perspektivy.

Obrázek 24: Ikony pro krokování kódu

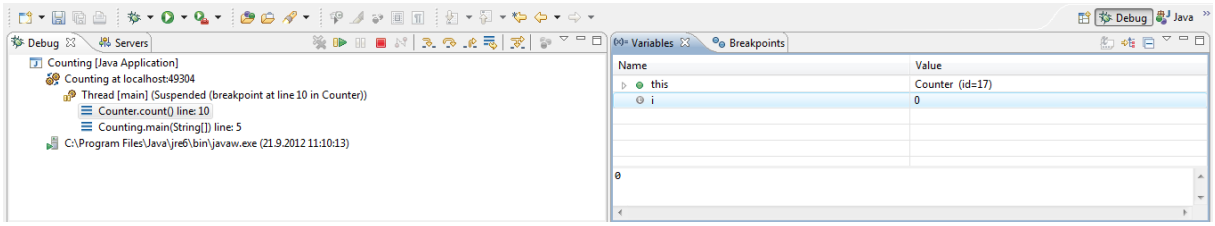
V části debugovací perspektivy nazvané *Debug* můžeme vidět aktuálně spuštěné části programu a také jejich vzájemný vztah (tyto aktuálně spuštěné části programu se nazývají angl. názvem **Stack**).

Obrázek 25: Okno Debug, kde můžeme vidět aktuální stack



Okno *Variables* zobrazuje pole a lokální proměnné z aktuálního stacku. Abychom mohli vidět toto okno, musí být spuštěný debugger.

Obrázek 26: Proměnné na aktuální stacku



Náš program se skládá ze dvou tříd, kde třída Counting využívá služeb třídy Counter. **Aktuální stack** je stack, který obsahuje všechny lokální vlastnosti třídy. Podíváme-li se na stack metody Counter.count() (viz. předchozí obrázek), vidíme vlastnosti třídy obsahující tuto metodu: lokální proměnnou i a proměnnou instance result. Klikneme-li však v okně Debug na metodu Counting.main(), vidíme v jejím stacku pole argumentů args a lokání proměnnou objekt counter. Přepínáním mezi různými stacky tak můžeme zjistit aktuální stav jakékoli proměnné a porovnávat ho s naším očekáváním. Pokud narazíme na rozdíl mezi naším očekáváním a aktuálním stavem proměnných, je možné, že jsme narazili na stopu pro odhalení nějaké chyby a musíme dále zkoumat, kde a proč se hodnoty proměnných odchýlily od námi očekávaných hodnot.

10.2 Výjimky

Další prostředek pro kontrolu správnosti chodu našeho kódu a odhalování případných chyb jsou tzv. **výjimky** (z angl. exceptions). Chybný průběh programu totiž nemusí nastat pouze v případech, kdy jsme z nějakého důvodu („Chybovat je lidské!“) napsali kód nesprávně, ale také v případech, kdy je náš kód napsán zcela správně – přesněji řečeno, náš program je napsán tak, aby správně pracoval ve všech očekávaných situacích, avšak mohou vzniknout situace neočekávané (např. v programu nepočítáme s variantou, že nebude fungovat připojení k Internetu), při nichž zpracovávaný program začne hlásit chyby.

Abychom program „vyladili“ i pro situace, kdy nastane nějaká výjimečná, neočekávaná či chybová záležitost, můžeme v Javě použít již výše zmíněné výjimky. Je trochu zavádějící uvažovat o výjimkách pouze v intencích chyb. Mechanismus výjimek pokrývá daleko širší prostor. V případě vzniku výjimky bychom měli chování kódu podřídit třem základním faktorům:

- úspěšně zpracovat vlastní výjimku,
- umožnit zpětné dohledání a analýzu výjimky,
- podat srozumitelnou informaci o výjimce navenek systému.

Zpracování vlastní výjimky je proces, kdy se v kódu rozhodujeme, jak s výjimkou naložit. Aniž bychom nyní zabíhali do rozdělení výjimek, ke kterému se dostaneme v další části textu, prozradíme si, že výjimku můžeme ošetřit anebo ji postoupit výše. Pokud máme v kontextu zachycení výjimky dost informací na její zpracování, měli bychom tak učinit.

Zpětné dohledání výjimky nastává v případech, kdy analyzujeme vznik výjimky, například při selhání systému nebo jeho neočekávaném chování. Činnost zpětného dohledání výjimky klade důraz na zaznamenání vlastní výjimky a poskytnutí maximálního počtu indicií, ze kterých je možné odvodit vznik výjimky.

Podáním srozumitelné informace se rozumí takové chování, kdy klient systému porozumí důvodu vzniku výjimky. Pokud předaná informace vyvolaná vznikem

výjimky inklinuje k variaci na téma zprávy Chyba:DataFormatException, dosáhneme spíše uživatelova zmatení než srozumitelného vysvětlení, že bylo zadáno datum v nesprávném formátu. Podání srozumitelné informace má více aspektů, například lokalizace chybové zprávy, vlastní forma zprávy (klient nemusí být uživatel, ale jiný systém) a podobně.

10.2.1 TRY, CATCH, FINALLY KONSTRUKCE A PŘÍKAZY THROW A THROWS

Před rozdělením výjimek si zopakujeme základní příkazy a konstrukce pro práci s výjimkami.

Chráněný blok, ošetření výjimky a finally

Chráněný blok se uvádí příkazem try a uzavřen mezi složené závorky. Jakoukoli výjimku vzniklou v chráněném bloku je možné ošetřit. K ošetření se používá příkaz catch, za nímž následuje konkrétní datový typ výjimky, kterou bude tento blok zpracovávat. Blok ošetření je opět omezen složenými závorkami.

```
try{
   //chráněný blok
}catch(Datovy typ výjimky){
   //ošetření výjimky
}
```

K chráněnému bloku patří i blok finally. Pokud vznikne v chráněném bloku výjimka, běh programu pokračuje nejdříve přes nejbližší blok catch, který ošetřuje výjimku daného datového typu, a poté přes blok finally (pokud je deklarován). Pokud výjimka nevznikne, běh programu pokračuje po posledním příkazu chráněného bloku opět přes blok finally. Blok finally se proto využívá pro operace, které musí být provedeny bez ohledu na fakt, jestli k výjimce dojde či nikoli - velmi často jde o uvolnění zdrojů (databázové připojení, soubor) alokovaných v rámci chráněného bloku.

```
try{
   //chráněný blok
}catch(Datovy typ výjimky){
   //ošetření výjimky
}finally{
   //kód, provedený bez ohledu na vznik výjimky
}
```

Je potřeba si uvědomit, že blok finally lze využít i bez bloku catch.

```
try{
   //chráněný blok
}finally{
   //kód, provedený bez ohledu na vznik výjimky
}
```

K výjimkám nerozlučně patří i příkazy throw a throws. Příkaz throw se používá k vyvolání výjimky. V podstatě existují dva způsoby - buďto konstruktorem vytvoříme novou výjimku a tu pomocí throw vyvoláme (throw new MyException();), nebo instanci výjimky již máme (například zachycením v bloku catch) a příkazem throw ji znovu vyvoláme.

Příkaz throws se zapisuje v signatuře metody a tvoří součást API public String doSomething() throws MyException(). Za throws následuje výčet datových typů výjimek, které může metoda vyvolat. Jakoukoli výjimku deklarovanou v signatuře není programátor nucen uvnitř metody ošetřovat. Naopak throws lze z hlediska klienta API považovat za zřeknutí se výjimky.

10.2.2 ZÁKLADNÍ ROZDĚLENÍ VÝJIMEK

Už víme, co je to výjimka a jak bychom na ni měli nahlížet. Nyní nastal čas seznámení s rozdělením výjimek. Java podporuje dva základní typy výjimek z hlediska jejich správy:

- **výjimky kontrolované**, takzvané checked exception
- **výjimky nekontrolované**, takzvané běhové či runtime exception

Kontrolované výjimky

Kontrolované výjimky jsou charakteristické tím, že klientský kód je nucen tyto výjimky ošetřit (catch blok) anebo propagovat (throws). Kontrola, jestli se tak děje, je provedena staticky, tedy v době kompilace kódu. Standardní API Javy deklaruje několik desítek kontrolovaných výjimek, které mohou nastat při práci s tou či onou částí API. Podívejme se například na ošetření a propagace výjimky vzniklé při práci se souborem. Ošetření kontrolované výjimky:

PŘÍKLAD 98

```
public void configure(final String confFileName){
   Properties props;
   try{
      File confFile = new File(confFileName);
      FileInputStream fis = new FileInputStream(confFile);
      props = new Properties();
      props.load(fis);
   }catch(FileNotFoundException fnf){
      //použijeme defaultní konfiguraci
      props = defaultProps;
   }
   ...
   ...
   ...
}
```

A propagování kontrolované výjimky:

PŘÍKLAD 99

```
public void configure(final String confFileName) throws FileNotFoundException{
   File confFile = new File(confFileName)
   FileInputStream fis = new FileInputStream(confFile);
   Properties props = new Properties();
   ...
```

```
    ...
    ...
}
```

Nekontrolované výjimky

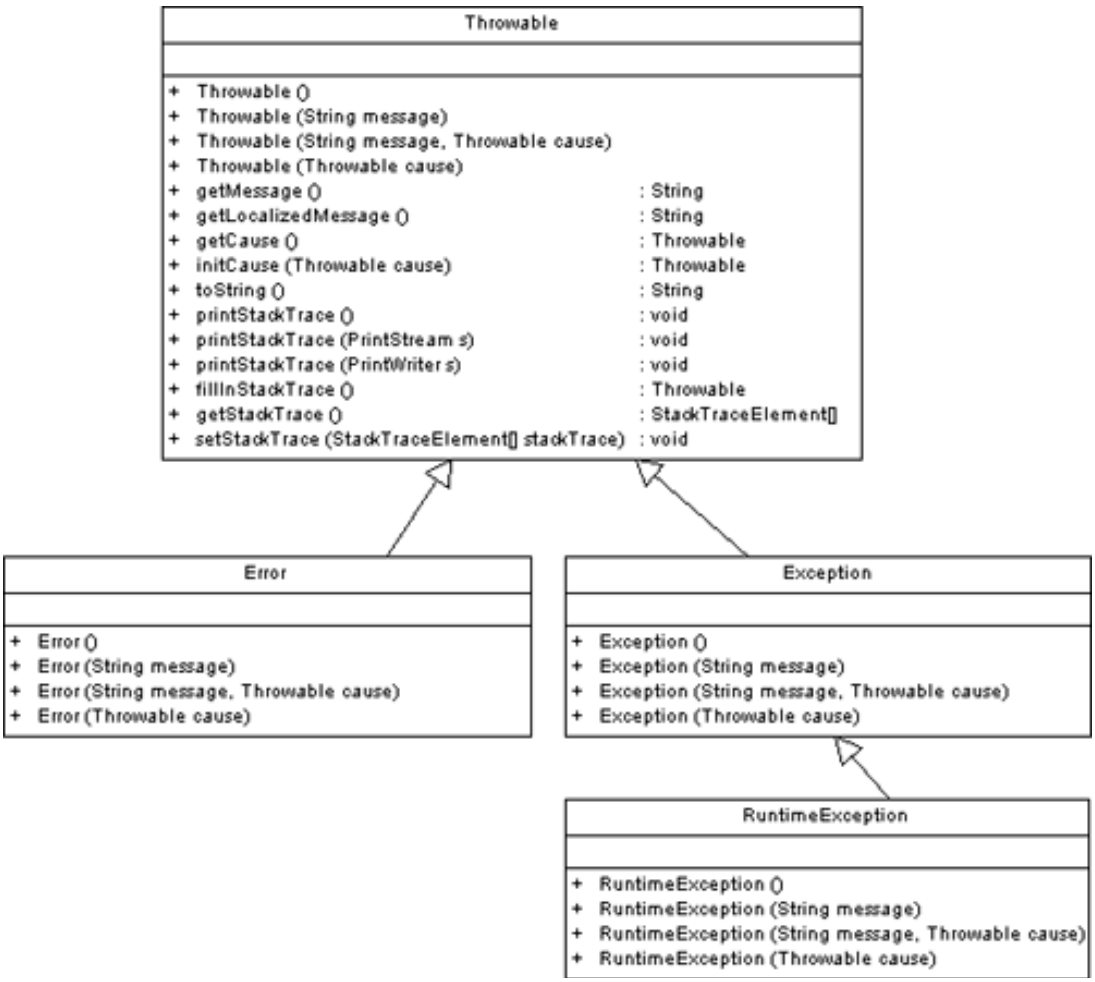
Nekontrolované či běhové výjimky se od kontrolovaných odlišují v tom, že programátor není explicitně nucen k jejich ošetření či propagaci. Nekontrolované výjimky vznikají za běhu systému, a proto nemá cenu jejich výskyt predikovat a snažit se je ošetřit nebo je propagovat. Mezi tyto výjimky patří například práce mimo rozsah pole ArrayIndex OutOfBoundsException, práce na null referenci NullPointerException a podobně.

10.2.3 OBJEKTOVÁ HIERARCHIE VÝJIMEK

Kořenovou třídou pro všechny výjimky a chyby představuje Throwable. Standardní API Javy definuje dva potomky, Error a Exception. Třída Error a její potomci představují rodinu závažných chyb, jedná se o chyby nekontrolované (běhové) a jejich chytání nebo převod na kontrolované nemá smysl.

Do kategorie závažných chyb spadají například chyby vzniklé při běhu JVM (Java Virtual Machine), jako je nedostatek paměti, přetečení zásobníku nebo vnitřní chyba JVM. Třída Exception představuje základní třídu (předka) pro všechny výjimky. Od této třídy jsou přímo odvozeny všechny kontrolované výjimky. Nekontrolované výjimky jsou odvozeny od RuntimeException. Základní hierarchii znázorňuje následující model tříd:

Obrázek 27: Model tříd výjimek



Takto nastavená hierarchie se využívá při tvorbě odvozených výjimek, které z těchto základních vycházejí. Výjimky mají ještě jednu základní vlastnost, umožňují řetězení, takzvaný **exception chaining**, o kterém se dozvíme na konci následující podkapitoly.

10.2.4 „POŽÍRÁNÍ“ VÝJIMEK

Pohlcení výjimky bez jakéhokoli zpracování patří mezi základní chyby, kterých bychom se měli vyvarovat. Snad nejhorší možnost nastane v případě, kdy deklarujeme chycení výjimky na úrovni předka (Exception). Zvažme následující příklad:

PŘÍKLAD 100

```
public class Account{
   ...
   ...//business logika
   ...
   public long getAccountBalance(){
      long balance = 0;
      try{
         .
         //operace pro zjištění zůstatku
         .
```

```
        }catch(Exception e){}
        return balance;
    }
}
```

Máme třídu Account, která představuje účet. Třída má metodu getAccountBalance(), která vrací zůstatek na účtu. Zjištění zůstatku, implementované v metodě getAccountBalance(), je uzavřeno v chráněném bloku try. K tomuto chráněnému bloku náleží blok zpracování – catch, kde jsou všechny výjimky odvozené od Exception pohlceny, a to právě díky prázdnému tělu tohoto bloku.

Představme si příklad, kdy se zjištění zůstatku provádí operacemi nad databází. V určitou chvíli se přeruší spojení s databází, které vyvolá vznik výjimky. Takto vzniklá výjimka je zachycena naším prázdným catch blokem a metoda je ukončena vrácením nulového zůstatku. Klient se pak může vzteky potrhat, neboť systém se navenek chová zcela korektně.

Další nepříjemnou vlastností výše uvedeného kódu je, že případná výjimka není nikde zaznamenána. Nevíme tak, jestli se jedná skutečně o výjimku signalizující chybu systému, nebo o jiný typ výjimky, například že účet je zablokován.

Základní pravidla pro práci s výjimkami:

- Nikdy nepohlcujme výjimky.
- Porušíme-li pravidlo jedna, připomte vysvětlující komentář.

Pravidlo dva existuje pro výjimečné případy, které neznamenají chybu. Základní API Javy například definuje výjimku InterruptedException, která se pojí s vlákny (vlákna umožňují pouštět více programů najednou, tzv. **multitasking**. Toto téma však spadá mimo záběr tohoto výukového materiálu) a jejich přerušením. I v takovém případě je dobrým zvykem vložit do prázdného catch bloku vysvětlující komentář.

10.2.5 OŠETŘOVÁNÍ VÝJIMEK NA ÚROVNI PŘEDKA

Ošetřování výjimek na úrovni předka (Exception) není závažné jako pohlcování výjimek, ale o to je zrádnější. Představme si kód, který je nucen zpracovat kontrolovanou výjimku. Tento případ ilustruje metoda fooBar, která volá metodu doSomethingFishy, jež deklaruje kontrolovanou výjimku:

PŘÍKLAD 101

```
public void doSomethingFishy() throws CheckedException{
    ...
    ...
}
```

PŘÍKLAD 102

```
public void fooBar(){
    doSomethingFishy();
    ...
    ...
}
```

Základní chybou je ošetřovat nebo propagovat výjimku CheckedException na úrovni předka, tedy Exception. Oba případy ilustrují následující ukázky.

PŘÍKLAD 103

```
public void fooBar(){
   try{
      doSomethingFishy();
   }catch(Exception e){
      //ošetření Exception
      ...
      ...
      ...
   }
}
```

V prvním případě jsme se sice rozhodli kontrolovanou výjimku ošetřit, ale učinili jsme tak na úrovni předka Exception. Na první pohled se to nemusí jevit jako chybné, ale je potřeba si uvědomit, že ošetřením na úrovni Exception zachytíme i všechny nekontrolované výjimky, které mohou signalizovat vážné selhání systému, například nenalezení odpovídající třídy (ClassNotFound).

Další úskalí představuje otázka, jak s takto obecnou výjimkou naložit. Jakým způsobem ji ošetřit? Máme v daném kontextu dostatek prostředků pro její zpracování? Většinou se v takovýchto případech použije pouhý výpis výjimky. Prostým „chytáním“ výjimky na úrovni Exception připravujeme aktuální kontext o možnost zotavení, neboť na výjimku můžeme nahlížet pouze jako na objekt Exception a nikoli jako na její generalizaci poskytující specifické informace. V případě propagace výjimky stavíme vyšší vrstvy před ten samý problém s obecnou (nekonkrétní) výjimkou.

Pokud je nezbytně nutné ošetřit výjimku na úrovni Exception, musíme oddělit výjimky kontrolované od nekontrolovatelných, tak jako v následující upravené ukázce:

PŘÍKLAD 104

```
public void fooBar(){
   try{
      doSomethingFishy();
   }catch(Exception e){
      //otestujeme jestli se jedná o běhovou výjimku
      if(e instanceof RuntimeException){
         //Běhovou výjimku necháme pokračovat
         throw (RuntimeException) e;
      }
      //zpracujeme všechny kontrolované výjimky
      ...
      ...
      ...
   }
}
```

Díky testu na běhovou výjimku jsme zajistili bezproblémový průchod těchto závažných výjimek. Na druhou stranu existují opodstatněné případy, kdy se na úrovni Exception výjimka chytá. Typickým příkladem je **Controller** webové aplikace, kdy je zapotřebí, nejlépe na centrálním místě, zpracovat všechny výjimky a

nabídnout uživateli rozumnou formou informaci o vzniku výjimky a umožnit mu tak další práci.

Výše uvedené řešení, s kontrolou na běhovou výjimku, lze přepsat pomocí deklarativního zpracování. Jedná se o to, že každý chráněný blok může mít několik bloků catch pro zpracování různých typů výjimek. V našem příkladu bychom upravili kód následujícím způsobem:

PŘÍKLAD 105

```
public void fooBar(){
   try{
      doSomethingFishy();
   }catch(RuntimeException re){
      throw re;//běhovou výjimku necháme pokračovat
   }catch(Exception e){
      //zpracujeme všechny kontrolované výjimky
      ...
      ...
      ...
   }
}
```

Nejdříve jsme deklarovali zpracování RuntimeException a teprve potom zpracování Exception. **Pořadí, v jakém jsme deklarovali jednotlivé typy výjimek, je důležité z hlediska zpracování.** Z předchozího textu již víme, že RuntimeException je potomek Exception. Pokud bychom deklarovali pořadí výjimek opačně, tedy nejdříve zpracování Exception, nedošlo by nikdy k zpracování (v našem případě opětovného vyhození běhové výjimky) RuntimeException. RuntimeException by skončila v prvním bloku, který jí odpovídá, což je blok s Exception. Obecně musíme mít na paměti, že nejdříve se v rámci několika catch bloků deklarují specializované typy výjimek a teprve poté se deklarují výjimky obecnějšího charakteru

10.2.6 PROPAGACE VÝJIMEK

PŘÍKLAD 106

```
public void fooBar() throws Exception{
   doSomethingFishy();
}
```

Druhý příklad, ve kterém jsme deklarovali vyhození výjimky Exception, je principielně špatný. Nutíme totiž všechny, jež volají metodu fooBar, aby výjimku zpracovali nebo propagovali. Odpovědnost za výjimku jsme alibisticky nechali na klientech využívajících API. Díky tomu, že vyhazujeme Exception, nutíme klienty vyhodit Exception a nebo ji ošetřit, aniž by k tomu měli prostředky.

Deklarace Exception se násobně pronáší do dalších API, neboť volající si není schopen s takto obecně definovanou výjimkou poradit. V konečném stadiu nastane veliký problém při práci s kontrolovanými výjimkami a klient bude nucen testovat, jestli je chycená výjimka určitého typu.

10.2.7 DEKLARACE NĚKOLIKA KONTROLOVANÝCH VÝJIMEK

Pokud jsme mluvili o deklarovaném vyhazování Exception, pak musíme zmínit druhý extrém, vyvolávání několika výjimek.

PŘÍKLAD 107

```
public void fooBar() throws CheckedExceptionX, CheckedExceptionY, CheckedExceptionZ{
   doSomethingFishy();
}
```

Programátor je nucen ošetřovat zbytečné množství výjimek. V takovýchto případech je vhodnější zvolit obecnější výjimku. Od této výjimky je pak možné vytvořit jednotlivé specializace na úrovní objektů anebo výjimku zřetězit do obecnější výjimky. Díky tomu můžeme v budoucnu ošetřit další typ výjimky, aniž by se to dotklo klientského programátora.

PŘÍKLAD 108

```
public class ApplicationException extends Exception{
    ...
    ...
    ...
}
public class ConfigApplicationException extends ApplicationException{
    ...
    ...
    ...
}
public class DataApplicationException extends ApplicationException{
    ...
    ...
    ...
}
public class Foo{
    private void configure() throws ConfigApplicationException{
        ...
        ...
        ...
    }
    private void postData() throws DataApplicationException{
        ...
        ...
        ...
    }
    public void fooBar() throws ApplicationException{
        configure();
        postData();
    }
}
```

10.2.8 ŘETĚZENÍ VÝJIMEK

Potřeba mechanismu **řetězení výjimek** (**exception chaining**) vyvstala postupně a jeho podpora se v API Javy objevila ve verzi 1.4. Řetězení výjimek vzniká vložením zachycené výjimky do výjimky jiné. Takto nově vytvořená výjimka se propaguje dále zcela standardním mechanismem. Díky tomuto přístupu nemusíme při rozšíření kódu propagovat nový typ výjimky. Zvažme následující kód:

PŘÍKLAD 109

```
public void doSomethingFishy() throws HighLevelException{
    try{
        ...
        ...
    }catch(LowLevelException e){
        throw new HighLevelException();
    }
}
```

Tento kód nedělá nic prostšího, než překlad kontrolované výjimky na jinou kontrolovanou výjimku, která je definována v rámci metody. Nevýhoda tohoto řešení je zřejmá, přijdeme o všechny důležité informace, které v sobě nese LowLevelException. Díky tomu už nemusí být dohledání příčiny vzniku HighLevelException snadné.

Mnohem lepší by bylo příčinu (LowLevelException) uschovat, respektive zřetězit do HighLevelException. Díky tomuto přístupu vyhovíme signatuře metody a zároveň neztratíme důležité informace o příčině vzniku HighLevelException. Následující kód ilustruje zřetězení výjimky:

PŘÍKLAD 110

```
public void doSomethingFishy() throws HighLevelException{
   try{
      ...
      ...
   }catch(LowLevelException e){
      throw new HighLevelException(e);
   }
}
```

Problematika řetězení výjimek, či výjimek obecně, je poměrně široká a zasahuje daleko za záběr tohoto výukového materiálu, tudíž zvědavější čtenáře můžeme pouze odkázat na další zdroje [21].

10.3 PŘÍKAZ ASSERT

Další možností pro ladění programu je použít klíčové slovo assert. Během ladění programu se snažíme obvykle na různá místa vkládat kontroly, zda platí určité předem dané podmínky. Například zda se nepokoušíme volat nějakou funkci s nesprávnou hodnotou argumentu. Tyto kontroly pak v okamžiku, kdy jsme zcela přesvědčeni o správnosti svého programu a zajímá nás již jen jeho rychlost, odstraníme nebo v lepším případě převedeme na komentáře. Brzy na to po objevení další chyby testy opět obnovujeme. Lepším řešením je použití příkazu assert, jenž umožňuje do přeloženého programu zařadit kontrolu, zda je splněna podmínka uvedená za klíčovým slovem assert. Pokud tato podmínka splněna není, vyvolá se výjimka AssertionError.

Testovanou podmínku můžeme doplnit po dvojtečce i textovým řetězcem, který se uloží jako popis výjimky. Při spuštění programu se pak v případě, že zadáme parametr *-ea*, provádí kontrola splnění zadaných podmínek. Pokud tento parametr nezadáme, kontrola podmínek se neprovádí a výpočet se tak ničím nezdržuje.

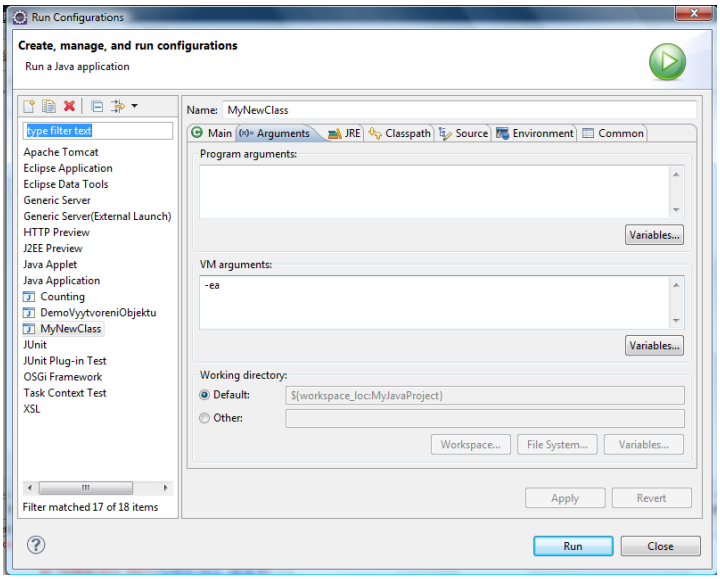
Definujme nyní funkci, která vrátí převrácenou hodnotu nenulového čísla, a ošetřeme situaci, kdy je jako argument předána nulová hodnota. S použitím příkazu assert bude řešení včetně testovacího programu následující:

PŘÍKLAD 111

```
class TestAssert {
  static double prevracena_hodnota(double x)
  {
    assert x != 0.0 : "Argument nesmi byt nulovy";
    return 1.0 / x;
  }
  public static void main(String args[]) {
    System.out.println(prevracena_hodnota(1));
    System.out.println(prevracena_hodnota(0));
  }
}
```

Program přeložíme a poté spustíme nejdříve bez parametru a poté s parametrem *–ea* (viz. Obrázek 28). Při prvním spuštění není ošetření výjimek zapnuto, takže se dělení nulou provede s výsledkem Infinity (nekonečno), zatímco při druhém spuštění je příkaz assert funkční a výpočet skončí výjimkou AssertionError.

Obrázek 28: Spuštění programu s parametrem *–ea*



Příkaz assert je vhodné uvádět na těch místech programu, kde potřebujeme zaručit platnost určité podmínky vyplývající přímo z logiky programu. Příkazem assert určitě nebudeme ošetřovat chyby vzniklé činností uživatele, neplatnost uvedené podmínky vždy indikuje chybu v našem programu (nebo chybně stanovenou podmínku, i na to si musíme dát pozor).

11 BALÍČKY

Balíčky se používají pro seskupování souvisejících objektů a souvisí s oblastí jmen. Jméno třídy odpovídá jménu souboru, ve kterém je definována třída. Jméno balíčku odpovídá adresáři, kde je daná třída umístěna. Bude–li například naše třída Kruznice umístěna v balíčku grafika.tvary, bude umístěna v souboru grafika\tvary\kruznice.class (zdrojový kód v grafika\tvary\kruznice.java). Pokud chcete tuto třídu v jiné třídě využívat, musí být v systémové proměnné CLASSPATH definována cesta k adresáři balíčku (adresář obsahující podadresář grafika), a na třídu se odkazujete pomocí tečkové notace, jak ukazuje příklad:

```
grafika.tvary.Kruznice k1 = new grafika.tvary.Kruznice();
```

Balíček třídy se definuje klíčovým slovem package umístěným jako první příkaz souboru. Příklad:

PŘÍKLAD 112

```
package grafika.tvary;
public class Osoba {
        ...
}
```

Pokud příkaz package neuvedeme, je třída součástí nepojmenovaného standardního balíčku. To se hodí pro malé aplikace nebo pro testovací účely, jinak raději balíčky používejme.

Standardní třídy jazyka Java jsou umístěny v podbalíčcích balíčku java: java.lang, java.util, java.awt.

```
java.lang.System.out.println(”Zdravime vlka!”);
```

Balíček java.lang obsahuje třídu System, ta má statickou proměnnou out (reference na objekt typu java.io.PrintStream) a ta má metodu println. Proměnná out třídy System představuje standardní výstup programu a metoda println vypíše řádek, celkově toto volání vypíše zadaný řetězec na standardní výstup. Jelikož balíček java.lang je základní balíček Javy, lze jeho jméno vynechávat a používat tak zkrácený zápis:

```
System.out.println(”Zdravime vlka!”);
```

Na okraj uvádíme, že Java rovněž umožňuje odkazovat se absolutně na třídy umístěné kdekoli na Internetu. K tomu se používá klasické doménové jméno. Předpokládejme, že chceme využívat třídu Circle umístěnou na serveru java.libraries.com v adresáři honza/grafika (http://java.libraries.com/honza/grafika/Circle.class). Kompletní jméno takové třídy by pak bylo COM.libraries.java.honza.grafika.Circle.

11.1 IMPORT BALÍČKŮ

Abysme nemuseli pokaždé vypisovat kompletní dlouhá jména všech tříd a jejich balíčků, umožňuje Java pomocí příkazu import určit, které třídy (balíčky) budeme

používat, a poté psát pouze zkrácený zápis. Pokud například často využíváme třídu grafika.tvary.Kruznice, stačí napsat za příkaz package před definici třídy import grafika.tvary.Kruznice;, a potom se na třídu lze odkazovat jako na Kruznice. Pokud používáme hodně tříd z balíčku java.awt (java.awt.Button, java.awt.List, java.awt.Checkbox, atd.), můžeme uvést příkaz import java.awt.*; a na třídy se odkazovat jen jako na Button, List, Checkbox, atd.

Příklad třídy:

PŘÍKLAD 113

```
package grafika.tvary;
import java.awt.*;
import java.applet.*;
import java.io.*;
import java.net.*;
public class MojeTrida {
        // definice třídy
        ...
        ...
}
```

11.2 JDK BALÍČKY

V souvislosti s balíčky se jedná o souvisejících skupinu tříd. Balíčky si můžeme vytvářet vlastní, Java však sama o sobě obsahuje množství „standardních“ balíčků. S rostoucí verzí se tento počet stále zvětšuje. Zatímco první verze Javy se skládala z pouhých 8 balíčků, verze 1.4 jich obsahuje 135. Zde si uvedeme přehled a popis těch nejdůležitějších z nich.

JDK 1.0 obsahovala těchto 8 balíčků: java.applet, java.awt, java.awt.image, java.awt.peer, java.io, java.lang, java.net a java.util. Verze JDK 1.1, kterou podporuje většina internetových prohlížečů, již obsahuje 22 balíčků. K původním přibyly například balíčky java.sql, java.math, java.awt.event nebo java.util.zip.

Následující tabulky by nám měly přinést komplexnější seznam nejdůležitějších balíčků s popisem a příkladem jejich jednotlivých tříd, rozhraní a výjimek.

Tabulka 13: JDK 1.0

Balíček	**Popis**
java.applet	Obsahuje třídy pro vytvoření appletů a pro komunikaci appletů se svým okolím. Jediná třída je Applet.
java.awt	Balíček pro definici uživatelského prostředí nezávislého na platformě (Abstract Window Toolkit). **Třídy**: Button (tlačítko), List (seznam), Font, Menu, Dialog, Point a interface LayoutManager...
java.awt.image	Obsahuje třídy pro práci s obrázky. **Třídy**: BufferedImage, ColorModel, ImageProducer, ImageConsumer, ImageFilter...
java.awt.peer	V balíčku jsou prvky uživatelského prostředí AWT pro konkrétní platformu (Windows, Solaris...). S touto třídou programátor běžně nepracuje!
java.io	Poskytuje třídy pro práci se vstupy a výstupy pomocí datových proudů a umožňuje práci se soubory. V programech ho není třeba importovat, je jako jediný balíček importován automaticky. **Nejdůležitější třídy**: Object, System, Thread, Throwable, Math, dále třídy pro datové typy Number, Integer, Short, Byte, Boolean, Character, Float, String... **Rozhraní**: např. Cloneable, Runnable. **Výjimky**: Exception, NullPointerException, ArrayIndexOutOfBoundsException, ClassNotFoundException, NumberFormatException, RuntimeException...
java.net	Obsahuje třídy umožňující komunikaci po Internetu (stažení dokumentu apod.). **Třídy**: URL, Socket, InetAddress, URLConnection, HttpURLConnection, URLEncoder... **Výjimky**: ConnectException, MalformedURLException, ProtocolException, SocketException...
java.util	Rozmanitý soubor utilit a datových struktur, obsahuje mimo jiné třídy pro reprezentaci data a času, generátor náhodných čísel a základní třídu pro událost. **Třídy**: Vector (rozšiřitelné pole), Random (generátor náhodných čísel), Date (implementuje časový okamžik), TimeZone (práce s časovými zónami), BitSet (rozšiřitelné bitové pole), EventObject (základní třída, od které všechny události dědí), Hashtable (hašovací tabulka)...

Tabulka 11-2: JDK 1.1

Balíček	Popis
java.awt.event	Obsahuje mechanismy pro vyvolání a zpracování událostí. **Třídy**: ActionEvent, FocusEvent, ItemEvent, MouseAdapter, MouseEvent, WindowEvent ... **Rozhraní**: ActionListener, FocusListener, ItemListener, KeyListener, MouseListener ...
java.beans	Slouží pro vývoj uživatelských komponent JavaBeans.
java.rmi	Slouží pro vytváření distribuovaných aplikací pomocí technologie RMI (Remote Method Invocation).
java.sql	Poskytuje třídy pro práci s (relačními) databázovými servery pomocí technologie JDBC (Java Database Connectivity).
java.util.zip	Poskytuje třídy pro čtení a zápis souborů formátu ZIP a GZIP. Třídy: ZipFile, ZipEntry, ZipInputStream, ZipOutputStream.

Jak bylo řečeno v úvodu, množství balíčků v současných verzích je velmi značné. Při běžném programování samozřejmě většinu těchto balíčků nevyužijeme. Proto o dalších už jen velmi stručně. V novějších verzích přibyly například:

- balíčky pro Swing – modifikovatelné uživatelské prostředí kvalitnější a uživatelsky příjemnější než AWT;
- balíčky pro podporu technologie Drag and Drop;
- balíček pro rychlé 2D geometrické operace;
- balíčky pro práci se šiframi a balíčky implementující kryptografické algoritmy;
- balíčky pro distribuované programování pomocí technologie CORBA;
- balíčky pro práci se zvukem;
- balíčky pro parsing XML dokumentů (SAX i DOM parsery)
- balíčky pro práci s regulárními výrazy.

O růstu jazyka Java svědčí i počet tříd v jednotlivých verzích. Zatímco JDK 1.0 obsahovala 211 tříd a rozhraní a verze JDK 1.1 pak 477 tříd a rozhraní, JDK 1.4 jich obsahuje již 2723.

11.3 Co je JVM a JIT?

Java Virtual Machine (JVM) je software, který funguje jako rozhraní mezi programem v jazyce Java, který byl přeložen do instrukcí tzv. "bytekódu", jemuž JVM rozumí, a procesorem s operačním systémem, který skutečně provádí instrukce po té, co je bytekód "za chodu" prostřednictvím JVM konvertován do nativních (binárních) instrukcí. Jakmile je pro danou platformu procesoru s operačním systémem k dispozici JVM a knihovny aplikačního rozhraní Java API (například API pro I/O operace), jakýkoli javovský program zkompilovaný do bytekódových třídních souborů (tzv. class files **JAR**), může být na JVM spuštěn. Musí však používat rozhraní Java API.

Výstup "kompilace" zdrojového javovského programu se nazývá **Java bytekód**. Sestává z bytekódových třídních souborů shromážděných v JAR (v Java ARchivu) a z bytekódových metod v každé javovské třídě souborů (což jsou spustitelné jednotky každé třídy souborů). JVM může buď interpretovat bytekódy jednotlivých

bytekódových metod jeden bytekód za druhým, mapujíc bytekódy do jedné nebo více hardwarových instrukcí cílového procesoru, jež jsou okamžitě provedeny. Nebo lze všechny bytekódy vybraných metod (například metody vyvolané uvnitř smyčky) dále přeložit "za chodu" tzv. "just-in-time" (JIT) kompilátorem do nativních kódových metod, které jsou vyvolány a spuštěny, když je vyvolána a spuštěna původní bytekódová metoda. To má za následek značné urychlení většiny programů v jazyce Java, zejména těch, jejichž provádění vyžaduje vícenásobné vyvolání zkompilovaných ("JITted") bytekódových metod.

12 PŘÍPADOVÁ STUDIE

Tato poslední kapitola je koncipována jako vrchol našeho snažení, kde si shrneme téměř všechny dosud získané poznatky na podrobněji rozpracovaném příkladu.

Představme si, že jsme zaměstnanci firmy, která dostala od Obchodně podnikatelské fakulty Slezské Univerzity (OPF SU) zakázku na zpracování informačního systému (IS) nazvaného *Univerzita*, který by spravoval informace o kantorech a studentech na této fakultě. Zadání pro objednaný IS by znělo takto:

IS *Univerzita* by měl zvládat tyto úkony:

1. Vkládání jednotlivých kantorů do IS – je-li kantor zároveň šéf katedry, bude IS taktéž zaznamenávat, kdo jsou jeho přímí podřízení
2. Vkládání jednotlivých studentů do IS
3. Jak kantor, tak student, bude mít v IS zaznamenáno jméno, e-mail a bydliště.
4. Vkládání předmětů vyučovaných na OPF SU do IS
5. Přiřazení jednotlivých předmětů ke kantorům
6. Zapisování studentů na předměty
7. Zrušení zápisu předmětů pro jednotlivé studenty
8. Zapisování známek studentům u jejich zapsaných předmětů
9. Výpis všech kantorů – pokud je kantor šéf katedry, bude při výpisu zvýrazněn. Dále bude systém umožňovat výpis počtu předmětů, které učí daný kantor.
10. Výpis všech studentů společně – u každého budou vypsány všechny předměty, které student studuje, společně s jejich známkami, celkový počet studovaných předmětů a průměrná známka z tohoto předmětu
11. Výpis všech předmětů - u každého předmětu bude uveden kantor, který tento předmět vyučuje

IS *Univerzita* bude zpracována dle objektového návrhu v JavaSE v prostřední Eclipse Indigo Sr2. K IS bude dodán podrobný manuál s názornými ukázkami použití všech požadovaných funkcí.

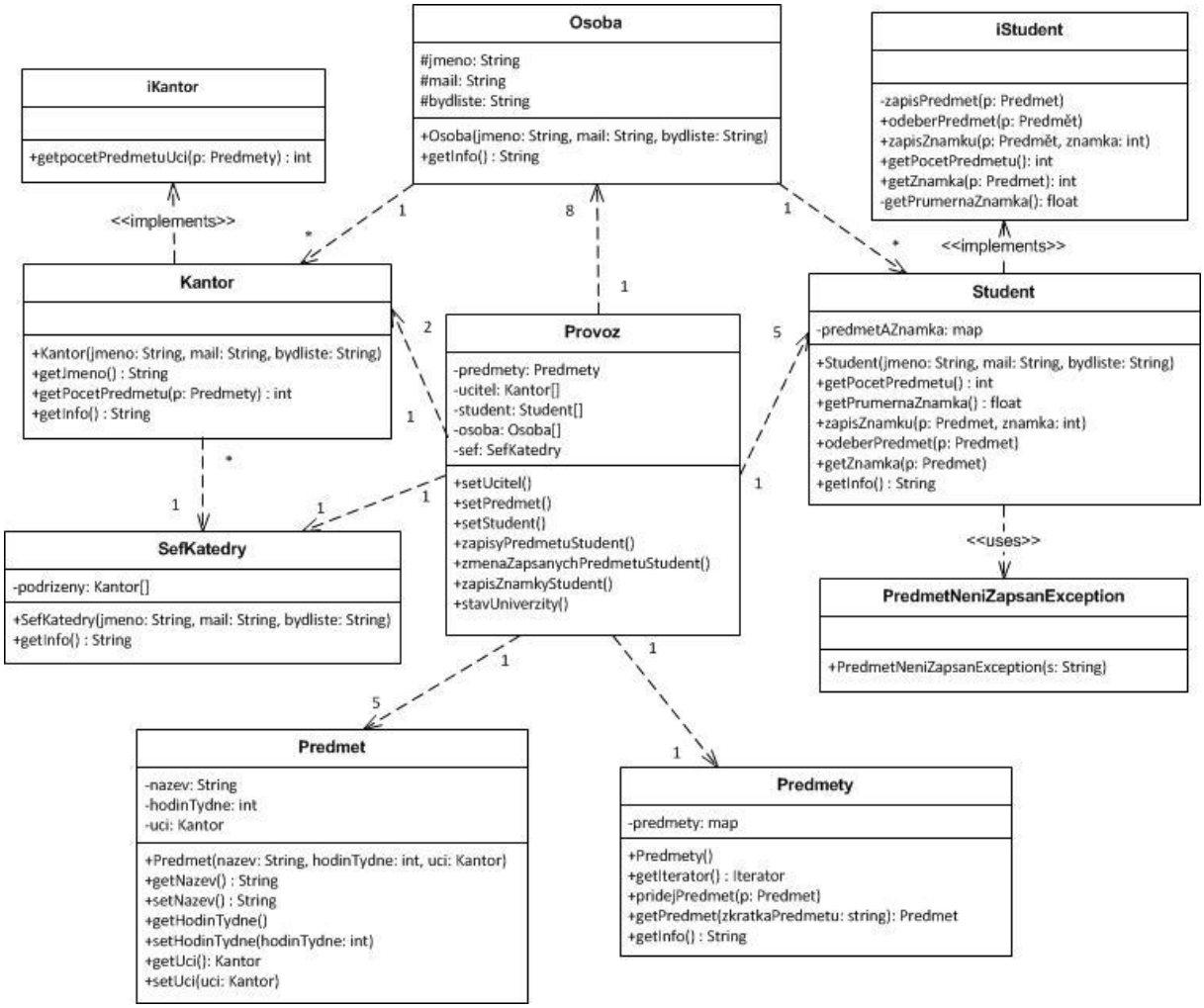
Po obdržení tohoto zadání si ho nejprve detailně prostudujeme a potom se (nejlépe s tužkou a papírem) pustíme nejdříve do hrubého návrhu IS (viz. Obrázek 12-1).

Nejdříve indetifikujeme všechny objekty, které se v zadání pro IS vyskytují. Jde o objekty *Kantor*, *Šéf Katedry*, *Student* a *Předmět*. Každý z těchto typů objektů by mohl mít jednotou šablonu ve formě třídy – v našem projektu bychom tedy logicky definovali třídy Kantor, SefKaterdy, Student a Předmět. Než však podrobně začneme identifikovat stavy a akce těchto reálných objektů, resp. vlastnosti a metody jejich softwarových ekvivalentů, zabývejme se nejdříve otázkou, zda některé z objektů nejsou pouze speciálními typy jiných, popř. zda některé objekty nesdílejí značné množství vlastností a bylo by vhodné tyto společné vlastnosti popsat nějakým společným typem objektu a tyto podobné objekty potom ze společného odvodit.

Na první pohled můžeme vidět, že *Šéf Katedry* je pouze speciální případ *Kantora*, který navíc obsahuje informaci o tom, kdo jsou jeho podřízení. V rámci objektově orientovaného návrhu bychom tedy definovali třídu SefKaterdy jako

podtřídu třídy Kantor. Další věc, které si všimneme, je podobnost *Kantora*, *Šéfa Katedry* a *Studenta* v tom smyslu, že u všech bude IS zaznamenávat shodné informace o jejich stavu – *jméno*, *e-mail* a *bydliště*. Bylo by jistě zbytečné definovat u každé třídy stejné vlastnosti jmeno, mail, bydliste. Mnohem elegantnější řešení je vytvořit pro třídy Kantor, ŠéfKaterdy a Student společnou nadtřídu, např. Osoba, která bude tyto informace jednotně uchovávat, a její potomci budou k těmto vlastnostem přistupovat skrze dědičnost.

Obrázek 12-1: Diagram tříd



Můžeme tedy rovnou navrhnout kód třídy Osoba:

PŘÍKLAD 114

```
public class Osoba {

	protected String jmeno;
	protected String mail;
	protected String bydliste;

	public Osoba(String jmeno, String mail, String bydliste) {
		this.jmeno = jmeno;
		this.bydliste = bydliste;
		this.mail = mail;
	}

	public String getInfo() {
		return "Jmeno: " + jmeno + "  e-mail: " + mail + "  bydliste: " + bydliste;
	}

}
```

Abychom v podtřídách třídy Osoba mohli přímo přistupovat k jednotlivým vlastnostem, nastavili jsme modifikátor přístupu na chráněný (protected). Kromě zmíněných vlastností a konstruktoru jsme ještě přidali požadovanou metodu nazvanou getInfo() pro výpis těchto třech vlastností.

Nyní bychom se zaměřili na vlastnosti jednotlivých tříd. Začněmě třídou Kantor. Po této třídě budeme požadovat, aby uměla vrátit počet předmětů, který daná instance (konkrétní kantor) vyučuje. Tuto metodu nazveme např. getPocetPredmetuUci(). Jelikož však zatím neznáme její implementaci a víme pouze to, že tuto metodu budeme po třídě Kantor požadovat, vložíme tuto třídu do rozhraní, které nazveme např. iKantor. Rozhraní, které bude vypadat takto,

PŘÍKLAD 115

```
public interface iKantor {
	public int getPocetPredmetuUci(Predmety p);
}
```

bude poté implementovat ve třídě Kantor. Hlavička třídy Kantor tedy bude vypadat následovně:

public class Kantor extends Osoba implements iKantor

Nyní se věnujme třídě SefKatedry. Tato třída bude potomkem třídy Kantor, která navíc bude nést informaci o podřízených. Implementace této metody může vypadat takto:

PŘÍKLAD 116

```
public class SefKaterdy extends Kantor {

	private Kantor[] podrizeni;

	public SefKaterdy(String jmeno, String mail, String bydliste, Kantor[] podrizeni) {
		super(jmeno, mail, bydliste);
		this.podrizeni = podrizeni;
	}

	public String getInfo() {
		String KantorInfo = super.getInfo();
		String sefInfo = "\nPodrizeni:\n";

		for (int i = 0; i < podrizeni.length; i++) {
			sefInfo += i + ": "  + podrizeni[i].getJmeno() + "\n";
		}
		return KantorInfo + sefInfo;
	}

}
```

Podřízení jsou uloženi v poli podrizeni typu Kantor (zvýrazněno tučně). V konstruktoru přidáme nastavení této speciální vlastnosti (oproti třídě Kantor) řádkem this.podrizeni = podrizeni; a metodu getInfo() překryjeme novou verzí, která navíc všechny podřízené vypíše. Všiměme si (tučně zvýrazněno v příkladu), že v této metodě využíváme výsledek překryté metody. Zde spočívá velká síla OOP, kdy není třeba psát ani kopírovat stejný kód na různá místa v programu, ale stačí využít již předtím definovanou metodu předka. Pokud bychom navíc později chtěli zaznamenávat u všech osob navíc informaci např. o věku, proměnnou věk přidáme pouze do třídy Osoba, upravíme její metodu getInfo(), tak aby se vek vypisoval, ale v kódu podtříd, které tuto proměnnou dědí, se tato změna vůbec neprojeví.

Nyní se věnujme třídě Student. Po této třídě požadujeme, aby nabízela metody, které:

- studentovi provedou zápis předmětu,
- studentovi odeberou zapsaný předmět,
- studentovi provedou zápis známky u zapsaného předmětu,
- vrátí počet zapsaných předmětů u daného studenta,
- vrátí známku pro daný předmět u daného studenta,
- vrátí průměrnou známku ze všech ohodnocených předmětů u daného studenta.

Tento požadavek, stejně jako u třídy Kantor, vede na tvorbu rozhraní iStudent, které můžeme implementovat takto:

PŘÍKLAD 117

```
public interface iStudent {
        void zapisPredmet(Predmet p);
        public void odeberPredmet(Predmet p) throws PredmetNeniZapsanException;
        public void zapisZnamku(Predmet p, int znamka) throws PredmetNeniZapsanException;

        public int getPocetPredmetu();
        public int getZnamka(Predmet p);  //znamka z konkretniho predmetu
        float getPrumernaZnamka();

}
```

Jelikož je možné, že v metodách odeberPredmet a zapisZnamku může dojít k výjimce, a to v případě, že se budeme snažit odebrat předmět, který není zapsán, popř. zapsat známku u předmětu, který není zapsán, přípíšeme k metodám konstrukci throws PredmetNeniZapsanException, která tuto chybu popisuje. Třídu výjimky může definovat takto:

PŘÍKLAD 118

```
public class PredmetNeniZapsanException extends Exception  {
        public PredmetNeniZapsanException(String s) {
                System.out.println(s);
        }
}
```

Třída výjimky nabízí pouze konstruktor, který vypíše hlášku (chybové hlášení) předávanou v parametru s.

V rozhraní iStudent se již objevuje třída Predmet. Zatím jsme tuto třídu pouze zmínili, nyní bychom se však mohli zabývat jejími vlastnostmi. U každého předmětu budeme chtít znát tyto vlastnosti: název předmětu, počet hodin týdně a kdo tento předmět učí. Třídu tedy můžeme implementovat takto:

PŘÍKLAD 119

```
public class Predmet {
	private String nazev;
	private int hodinTydne;
	private Kantor uci;

	public Predmet(String nazev, int hodinTydne, Kantor uci) {
		this.nazev = nazev;
		this.hodinTydne = hodinTydne;
		this.uci = uci;
	}

	public String getNazev() {
		return nazev;
	}

	public void setNazev(String nazev) {
		this.nazev = nazev;
	}

	public int getHodinTydne() {
		return hodinTydne;
	}

	public void setHodinTydne(int hodinTydne) {
		this.hodinTydne = hodinTydne;
	}

	public Kantor getUci() {
		return uci;
	}

	public void setUci(Kantor uci) {
		this.uci = uci;
	}
}
```

Je zřejmé, že tato třída slouží pouze jako úložiště informací o předmětu.

Kde však uložíme všechny předměty dohromady? Můžeme použít pole, ale vzhledem k tomu, že můžeme potřebovat seznam předmětů navyšovat, či nějaké odebírat, bude vhodnější nějaký kontejner, např. HashMap. Předměty pak můžeme ukládat do vlastnosti predmety třídy Predmety, která by mohla být implementována takto:

PŘÍKLAD 120

```
import java.util.*;

public class Predmety {

	private HashMap predmety;

	public Predmety() {
		predmety = new HashMap();
	}

	public Iterator getIterator() {
		return predmety.values().iterator();
	}

	public void pridejPredmet(Predmet p) {
		predmety.put(p.getNazev(), p);
	}

	public Predmet getPredmet(String zkratkaPredmetu) {
		return (Predmet)predmety.get(zkratkaPredmetu);
	}

	public String getInfo() {
		String uciPredmety = "Vyuka predmetu:\n";
		Set set = predmety.entrySet();
		Iterator i = set.iterator();
		while(i.hasNext()) {
			Map.Entry me = (Map.Entry)i.next();
			uciPredmety +=  (String)me.getKey() + ": " +
((Predmet)me.getValue()).getUci().getJmeno() + "\n";
		}
		return uciPredmety;
	}
}
```

Je zřejmé, k čemu jsou jednotlivé metody, jediná nejasnost může být u metody getIterator(). Význam této metody vyplyne z implementace nejsložitější třídy, což je třída Student a dále z implementace třídy Kantor. Implementace třídy Student by mohla vypadat takto:

PŘÍKLAD 121

```
import java.text.NumberFormat;
import java.util.*;

public class Student extends Osoba implements iStudent {

	private Map predmetAZnamka;

	public Student(String jmeno, String mail, String bydliste) {
		super(jmeno, mail, bydliste);
		predmetAZnamka = new HashMap();
		// TODO Auto-generated constructor stub
	}

	@Override
	public int getPocetPredmetu() {
		return predmetAZnamka.size();
	}

	@Override
	public float getPrumernaZnamka() {
		int sumaZnamek = 0;
		Set set = predmetAZnamka.entrySet();
		// Get an iterator
		Iterator i = set.iterator();
		// Display elements
		while(i.hasNext()) {
			Map.Entry me = (Map.Entry)i.next();
			sumaZnamek += ((Integer)me.getValue()).intValue();
		}
		return (float)sumaZnamek/getPocetPredmetu();

	}

	@Override
	public void zapisPredmet(Predmet p) {
		predmetAZnamka.put(p, new String("Zadna"));
		// TODO Auto-generated method stub
	}

	public void zapisZnamku(Predmet p, int znamka) throws PredmetNeniZapsanException{
		if (predmetAZnamka.containsKey(p)) {
			predmetAZnamka.remove(p);
			predmetAZnamka.put(p, new Integer(znamka));
		} else {
			throw new PredmetNeniZapsanException("Znamka nemuze byt zapsana:
student " + jmeno + " nema predmet zapsan.");
		}
	}

	@Override
	public void odeberPredmet(Predmet p) throws PredmetNeniZapsanException {
		if (predmetAZnamka.containsKey(p)) {
			predmetAZnamka.remove(p);
		} else {
			throw new PredmetNeniZapsanException("Predmet nemuze byt odebran:
student " + jmeno + " nema predmet zapsan.");
```

```
		}
	}

	@Override
	public int getZnamka(Predmet p) {
		if (predmetAZnamka.containsKey(p)) {
			return ((Integer)predmetAZnamka.get(p)).intValue();
		} else {
			System.out.println("Predmet neni zapsan.");
			return 0;
		}
	}

	@Override
	public String getInfo() {
		String zaklInfo = super.getInfo();

		String znamkyZPredmetu = "";
		Set set = predmetAZnamka.entrySet();
		Iterator i = set.iterator();
		while(i.hasNext()) {
			Map.Entry me = (Map.Entry)i.next();
			znamkyZPredmetu +=  ((Predmet)me.getKey()).getNazev() + ": " +
((Integer)me.getValue()).intValue() + "  ";
		}
		znamkyZPredmetu += "\n";

		String prumernaZnamka =
NumberFormat.getInstance().format(getPrumernaZnamka());
		String out = "Student:\n" + zaklInfo + "\n" + "Znamky z predmetu: " +
znamkyZPredmetu;
		return out + "Pocet predmetu: " + getPocetPredmetu() + "  Prumerna znamka: " +
prumernaZnamka;
	}

}
```

Tato třída obsahuje vlastnost predmetAZnamka typu HashMap, kde jsou pro danou instanci (konkrétního studenta) ukládány informace o tom, které předměty si zapsal a jakou z nich dostal známku. Význam dalších metod je zřejmý.

Nyní zpracujeme poslední zmíněnou třídu, jejíž implementaci jsme se ještě nevěnovali. Jedná se o třídu Kantor a její implementace může vypadat takto:

PŘÍKLAD 122

```
import java.util.*;

public class Kantor extends Osoba implements iKantor {

	public Kantor(String jmeno, String mail, String bydliste) {
		super(jmeno, mail, bydliste);
		// TODO Auto-generated constructor stub
	}

	public String getJmeno() {
		return jmeno;
	}
```

```
	public int getPocetPredmetuUci(Predmety p) {
		int pocet = 0;
		Iterator it = p.getIterator();
		while(it.hasNext()) {
			if (this == ((Predmet)it.next()).getUci()) {
				pocet++;
			}
		}
		return pocet;
	}

	@Override
	public String getInfo() {
		return "Kantor:\n" + super.getInfo();
	}
}
```

Implementace všech tříd potřebných k provozu je tedy hotova. Můžeme ji otestovat pomocí následující spustitelné třídy Provoz:

PŘÍKLAD 123

```
public class Provoz {

	/**
	 * @param args
	 */
	static Predmety predmety;
	static Kantor[] Kantor;
	static Student[] student;
	static Osoba[] osoba;
	static SefKaterdy sef;

	static void setKantor() {
		Kantor = new Kantor[2];
		Kantor[0] = new Kantor("Jan Gorecki", "gorecki@opf.slu.cz", "Karvina");
		Kantor[1] = new Kantor("Roman Sperka", "sperka@opf.slu.cz", "Cadca");

		sef = new SefKaterdy("Dominik Vymetal", "vymetal@opf.slu.cz", "Hradek", Kantor);
	}

	static void setPredmet() {
		predmety.pridejPredmet(new Predmet("ZOP", 4, Kantor[1]));
		predmety.pridejPredmet(new Predmet("BPZIE", 4, Kantor[1]));
		predmety.pridejPredmet(new Predmet("PZNI", 2, Kantor[0]));
		predmety.pridejPredmet(new Predmet("PPSA", 3, Kantor[0]));
		predmety.pridejPredmet(new Predmet("PMES", 3, sef));
	}

	static void setStudent() {
		student = new Student[5];
		student[0] = new Student("Jiri Novak", "o123456@opf.slu.cz", "Brno");
		student[1] = new Student("Pavel Viril", "o100000@opf.slu.cz", "Krnov");
		student[2] = new Student("Radim Urch", "o110011@opf.slu.cz", "Karvina");
		student[3] = new Student("Peter Vodnar", "o121212@opf.slu.cz", "Zilina");
		student[4] = new Student("Jaromir Rak", "o111111@opf.slu.cz", "Praha");
	}

	static void zapisyPredmetuStudent() {
```

```
		//tretaci
		student[0].zapisPredmet(predmety.getPredmet("ZOP"));
		student[0].zapisPredmet(predmety.getPredmet("PPSA"));

		student[1].zapisPredmet(predmety.getPredmet("ZOP"));
		student[1].zapisPredmet(predmety.getPredmet("PPSA"));

		//patak
		student[2].zapisPredmet(predmety.getPredmet("PZNI"));

		//vecni studenti
		student[3].zapisPredmet(predmety.getPredmet("ZOP"));
		student[3].zapisPredmet(predmety.getPredmet("PPSA"));
		student[3].zapisPredmet(predmety.getPredmet("BPZIE"));

		student[4].zapisPredmet(predmety.getPredmet("ZOP"));
		student[4].zapisPredmet(predmety.getPredmet("PPSA"));
		student[4].zapisPredmet(predmety.getPredmet("BPZIE"));
		student[4].zapisPredmet(predmety.getPredmet("PZNI"));
	}

	static void zmenaZapsanychPredmetuStudent() throws PredmetNeniZapsanException {
		student[4].odeberPredmet(predmety.getPredmet("ZOP"));

		student[4].zapisPredmet(predmety.getPredmet("PMES"));
	}

	static void zapisZnamkyStudent() throws PredmetNeniZapsanException {
		student[0].zapisZnamku(predmety.getPredmet("ZOP"), 3);
		student[0].zapisZnamku(predmety.getPredmet("PPSA"), 2);

		student[1].zapisZnamku(predmety.getPredmet("ZOP"), 2);
		student[1].zapisZnamku(predmety.getPredmet("PPSA"), 2);

		//patak
		student[2].zapisZnamku(predmety.getPredmet("PZNI"), 1);

		//vecni studenti
		student[3].zapisZnamku(predmety.getPredmet("ZOP"), 4);
		student[3].zapisZnamku(predmety.getPredmet("PPSA"), 3);
		student[3].zapisZnamku(predmety.getPredmet("BPZIE"), 3);

		student[4].zapisZnamku(predmety.getPredmet("PMES"), 3);
		student[4].zapisZnamku(predmety.getPredmet("PPSA"), 4);
		student[4].zapisZnamku(predmety.getPredmet("BPZIE"), 3);
		student[4].zapisZnamku(predmety.getPredmet("PZNI"), 4);
	}

	static void stavUniverzity() {
		osoba = new Osoba[8];

		//kopiruj vsechny osoby na univerzite do pole osoba
		System.arraycopy(Kantor, 0, osoba, 0, 2);
		osoba[2] = sef;
		System.arraycopy(student, 0, osoba, 3, 5);

		//vypis vsech osob
		for (int i = 0; i < 8; i++) {
			// sef katedry bude pri vypisu zvyraznen
			if (osoba[i] instanceof SefKaterdy) {
```

```
		System.out.println("****************************************************************");
				}
				System.out.print("Osoba: " + i + "  ");
				System.out.println(osoba[i].getInfo());
				if (osoba[i] instanceof SefKaterdy) {
					System.out.println("Uci predmetu: " +
((Kantor)osoba[i]).getPocetPredmetuUci(predmety) + "\n");

		System.out.println("****************************************************************");
				} else {
					System.out.println("");
				}

			}

			//vypis vsech predmetu a jejich vyucujicich
			System.out.println(predmety.getInfo());
		}

		public static void main(String[] args) {
			predmety = new Predmety();

			setKantor();
			setPredmet();
			setStudent();
			zapisyPredmetuStudent();
			try {
				zmenaZapsanychPredmetuStudent();
				zapisZnamkyStudent();
				stavUniverzity();
			} catch (PredmetNeniZapsanException e) {
				System.out.println("Doslo k chybe v systemu: stop vypisu.");
			} finally {
				System.out.println("\nKonec zpracovani.");
			}
		}
}
```

V tomto zkušebním příkladu jsme vyvořili a uložili do pole dvě instance třídy Kantor, jednu instanci třídy SefKatedry a pět instancí třídy Student (metody setKantor() a setStudent()). Dále jsme vytvořili seznam pěti předmětů, kde každému jsme přiřadili právě jednoho kantora (metoda setPredmet()). Studenti si poté zapsali předměty, provedli změnu v zápisech a potom jim byly zapsány známky (metody zmenaZapsanychPredmetuStudent() a zapisZnamkyStudent()). Nakonec jsou veškeré informace uložené v IS *Univerzita* vypsány skrze rozhraní třídy Osoba, které ukazuje další silnou stránku OOP – polymorfismus: se všemi rozdílnými osobami můžeme pracovat stejným způsobem u metod, které mají tyto osoby společné, v tomto případě tedy metodu getInfo(). Výsledkem výpisu po průběhu metody main() u třídy Provoz tedy bude:

```
Osoba: 0  Kantor:
Jmeno: Jan Gorecki  e-mail: gorecki@opf.slu.cz  bydliste: Karvina

Osoba: 1  Kantor:
Jmeno: Roman Sperka  e-mail: sperka@opf.slu.cz  bydliste: Cadca
```

```
************************************************************
Osoba: 2  Kantor:
Jmeno: Dominik Vymetal  e-mail: vymetal@opf.slu.cz  bydliste: Hradek
Podrizeni:
0: Jan Gorecki
1: Roman Sperka

Uci predmetu: 1

************************************************************
Osoba: 3  Student:
Jmeno: Jiri Novak  e-mail: o123456@opf.slu.cz  bydliste: Brno
Znamky z predmetu: PPSA: 2  ZOP: 3
Pocet predmetu: 2  Prumerna znamka: 2,5

Osoba: 4  Student:
Jmeno: Pavel Viril  e-mail: o100000@opf.slu.cz  bydliste: Krnov
Znamky z predmetu: PPSA: 2  ZOP: 2
Pocet predmetu: 2  Prumerna znamka: 2

Osoba: 5  Student:
Jmeno: Radim Urch  e-mail: o110011@opf.slu.cz  bydliste: Karvina
Znamky z predmetu: PZNI: 1
Pocet predmetu: 1  Prumerna znamka: 1

Osoba: 6  Student:
Jmeno: Peter Vodnar  e-mail: o121212@opf.slu.cz  bydliste: Zilina
Znamky z predmetu: BPZIE: 3  PPSA: 3  ZOP: 4
Pocet predmetu: 3  Prumerna znamka: 3,33

Osoba: 7  Student:
Jmeno: Jaromir Rak  e-mail: o111111@opf.slu.cz  bydliste: Praha
Znamky z predmetu: BPZIE: 3  PPSA: 4  PZNI: 4  PMES: 3
Pocet predmetu: 4  Prumerna znamka: 3,5

Vyuka predmetu:
ZOP: Roman Sperka
BPZIE: Roman Sperka
PMES: Dominik Vymetal
PPSA: Jan Gorecki
PZNI: Jan Gorecki


Konec zpracovani.
```

Studiem tohoto výpisu se můžeme sami přesvědčit, že navržený IS *Univerzita* pracuje správně.

Nyní ještě zkontrolujme, zda IS reaguje správně v případě snahy o provedení chybné akce, např. když se pokusíme v provozu zapsat známku u předmětu, který nemá student zapsán. Nahraďme tučně zvýrazněný řádek ve třídě Provoz za tento řádek:

```
student[0].zapisZnamku(predmety.getPredmet("ZOPA"), 3);
```

Výpis po spuštění třídy Provoz bude vypadat takto:

```
Znamka nemuze byt zapsana: student Jiri Novak nema predmet zapsan.
Doslo k chybe v systemu: stop vypisu.

Konec zpracovani.
```

IS tedy reaguje na snahu o chybný zápis známky chybovým hlášením, díky kterému můžeme chybu v kódu třídy Provoz snadno najít a odstanit.

ZÁVĚR

Cílem učebního textu bylo seznámit čtenáře se základy objektově orientovaného programování. Úvodní části textu byly věnovány filosofii a paradigmatům OOP. Vzhledem k tomu, že všechny pojmy byly ilustrované praktickými příklady, bylo nutné si zvolit vhodné vývojové prostředí a programovací jazyk. Autoři vybrali prostředí Eclipse Indigo a jazyk Java, především kvůli jeho současné rozšířenosti a popularitě. Další části textu se proto věnují popisu zvoleného prostředí a jeho instalaci. Následuje vysvětlení základních prvků a příkazů jazyka Java. Největší prostor je věnován základním konstruktům programovacího jazyka, kterými jsou např. třídy, objekty, metody, konstruktory, destruktory a jiné. Závěrečné části textu objasňují problematiku dědičnosti, polymorfismu a balíčků. Poslední kapitola je komplexní případovou studií, která demonstruje využití poznatků, získaných z tohoto textu.

Učební text si v žádnem případě nečiní nároky na kompletní užívatelskou příručku. Pro pokročilejší znalosti OOP musí čtenář sáhnout po publikacích technologií OOP.

SEZNAM POUŽITÉ LITERATURY

[1] Apogee CZ, s.r.o. *Co je JVM a JIT?* [online]. 2011, 2011-07-20 [cit. 2012-09-21]. Dostupné z: http://www.apogee.cz/javacz.htm

[2] *Dynamické datové struktury* [online]. 2002 [cit. 2012-10-04]. Dostupné z: http://java.vse.cz/pdf/java-dynamicke_datove_struktury.pdf

[3] FARRELL, Joyce. Java programming. 6th Ed. Boston, MA: Course Technology, Cengage Learning, 2011, p. cm. ISBN 978-111-1529-444.

[4] FLANAGAN, David. Programování v jazyku JAVA: kompletní učebnice pro začátečníky. 1. vyd. Praha: Computer Press, 1997, 488 s. Myslíme v--. ISBN 80-858-9678-8.

[5] HATINA, Petr. *Programování v jazyku Java (1) - Úvod*. Linuxsoft.cz [online]. 2004, 2004-07-09 [cit. 2012-09-21]. Dostupné z: http://www.linuxsoft.cz/article.php?id_article=244

[6] *Java pro začátečníky: příručka vývojáře*. Algoritmy.net [online]. 2012 [cit. 2012-09-21]. Dostupné z: http://www.algoritmy.net/category/21338/Java-pro-zacatecniky

[7] LIANG, Daniel. *Introduction to JAVA programming: comprehensive version*. 8th ed. Boston: Prentice Hall, c2011, xxiv, 1342 p. ISBN 01-321-3080-7.

[8] NÁVRAT, Lumír. *Abstraktní třídy a metody*. In: Abstraktní třídy a metody [online]. [cit. 2012-10-04]. Dostupné z: http://www.cs.vsb.cz/navrat//vyuka/esf/java/ch01s05.html

[9] NETRVALOVÁ, Arnoštka. Jak na Eclipse. In: *Jak Na Eclipse* [online]. [cit. 2012-10-04]. Dostupné z: http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa1/portal/navody/eclipse-navod/eclipse.html

[10] NETRVALOVÁ, Arnoštka. Jak nainstalovat Javu. In: Jak nainstalovat Javu [online]. [cit. 2012-10-04]. Dostupné z: http://www.kiv.zcu.cz/~netrvalo/vyuka/ppa1/portal/navody /java-navod/java-instalace.html

[11] PECINOVSKÝ, Rudolf. *Myslíme objektově v jazyku Java: kompletní učebnice pro začátečníky*. 2., aktualiz. a rozš. vyd. Praha: Grada, 2009, 570 s. Myslíme v--. ISBN 978-80-247-2653-3.

[12] PICHLÍK, Roman. *Java a výjimky - úvod do problematiky výjimek*. Java a výjimky - úvod do problematiky výjimek [online]. 2005, 2005-01-27 [cit. 2012-10-04]. Dostupné z: http://interval.cz/clanky/java-a-vyjimky-uvod-do-problematiky-vyjimek/

[13] PROCHÁZKA, Jaroslav, VAJGL, Marek a Cyril KLIMEŠ. *Informační systémy 2: Učební text*. 2. vyd. Ostrava: Ostravská univerzita v Ostravě, 2010, 179 s.

[14] REGES, Stuart a Martin STEPP. *Building Java programs: a back to basics approach*. 2nd ed. Boston: Addison-Wesley, c2011, xxiii, 1151 p. ISBN 01-360-9181-4.

[15] SARANG, Poornachandra a Martin STEPP. Java programming: a back to basics approach. 2nd ed. New York: McGraw-Hill Professional, c2012, xxv, 642 p. ISBN 00-716-3360-X.

[16] SELVAGGIA. *Java - Vyjímky (31.díl)*. OWebu.cz [online]. 2008, č. 1, 2008-05-02 [cit. 2012-09-21]. Dostupné z: http://owebu.bloger.cz/Programovani/Java-Vyjimky-31-dil

[17] SEMECKÝ, Jiří. *Naučte se Javu - aplikace.* Interval.cz [online]. 2002, č. 1, 2002-08-30 [cit. 2012-09-21]. Dostupné z: http://interval.cz/clanky/naucte-se-javu-aplikace/

[18] SEMECKÝ, Jiří. *Naučte se Javu - balíčky*. Interval.cz [online]. 2002, č. 1, 2002-10-01 [cit. 2012-09-21]. Dostupné z: http://interval.cz/clanky/naucte-se-javu-balicky/

[19] SCHILDT, Herbert. *Java The Complete Reference guide*. 8th Ed. New York: McGraw-Hill Osborne Media, 2011, p. 1152. ISBN 0070435928.

[20] *Stručný úvod do jazyka JAVA*. COMPUTERWORLD. Gapo.cz [online]. 2012 [cit. 2012-09-21]. Dostupné z: http://gapo.cz/doc/docs/java/uvod_do_jazyka/index.html

[21] *The Java tutorials*. Oracle.com [online]. 2012 [cit. 2012-09-21]. Dostupné z: http://docs.oracle.com/javase/tutorial/

[22] VOGEL, Lars. Java Debugging with Eclipse - Tutorial. In: Java Debugging with Eclipse - Tutorial [online]. 2009, 2012-09-26 [cit. 2012-10-04]. Dostupné z: http://www.vogella.com/articles/EclipseDebugging/article.html

[23] ZICHA, Vojtěch. *Java tutoriál - Objekty a třídy (7. díl)*. Programujte.com: Java [online]. 2007, č. 1, 2007-08-27 [cit. 2012-09-21]. Dostupné z: http://programujte.com/clanek/2007082501-java-tutorial-objekty-a-tridy-7-dil/

[24] ZÍCHA, Vojtěch. Java tutoriál - Operátory (5. díl). Http://programujte.com [online]. 2007, č. 1, 2007-05-20 [cit. 2012-10-04]. Dostupné z: http://programujte.com/clanek/2007050401-java-tutorial-operatory-5-dil/

PŘÍLOHA Č. 1 : JAVA A DATABÁZE

Moderní aplikace ukládají provozní data do systému řízení báze dat (SŘBD), které bývají většinou relační, někdy ale také objektové či XML (eXtensible Markup Language). Mezi běžně používané relační databáze patří například Oracle, MS SQL, MySQL, DB2 a další. My se budeme zabývat pouze komunikací s relačními databázemi.

Pro práci s databází slouží v Javě balíček java.sql a javax.sql. Aplikační rozhraní (API – Application Programming Interface) pro přístup k datům v javě se jmenuje **JDBC** (Java Database Connectivity). Stejně jako všechny programy psané v Javě, je také JDBC platformě nezávislý. JDBC je součástí standardní Javy SE (od JDK verze 1.1) jako JSR 54. Další verze JDBC známé jako JSR 114 (obsahující Rowset) a JSR 221 jako JDBC 4.0 v Java SE 6.0.

JDBC je implementací rozhraní Driver. Pokud chceme tento ovladač využívat pro připojení k databázi, musíme jej nejdříve připojit k aplikaci. To provedeme registrací pomocí metody forName(String className) nebo přímou registrací driveru pomocí statické metody registerDriver() objektu java.sql.DriverManager. Pokud je balíček umístěn jinde, než je standardní lib adresář javy, je nutné cestu k jeho umístění přidat do CLASSPATH. JDBC definuje čtyři typy ovladačů, jedná se o:

1. JDBC-ODBC most – pomalé připojení, navíc nutnost využití ODBC ovladače na straně klienta.
2. Ovladač napsaný v nativním API, používá nativní knihovny.
3. Ovladač používá čistého Java klienta, komunikace probíhá pomocí nezávislého protokolu, data jsou pomocí něj vyžadována.
4. Ovladač je čistá Java, komunikace probíhá pomocí specifického protokolu, připojení existuje přímo na datový zdroj.

Vlastní připojení k databázi potom probíhá pomocí statické metody getConnection() třídy DriverManager, která:

- obsahuje jako parametr většinou URL databáze, jméno a heslo pro přístup do databáze (k danému schématu),
- vrací instanci Connection,
- v případě neúspěšného navázání spojení vyhazuje SQLException.

PŘÍKLAD 124 : MYSQL PŘÍKLAD PŘIPOJENÍ

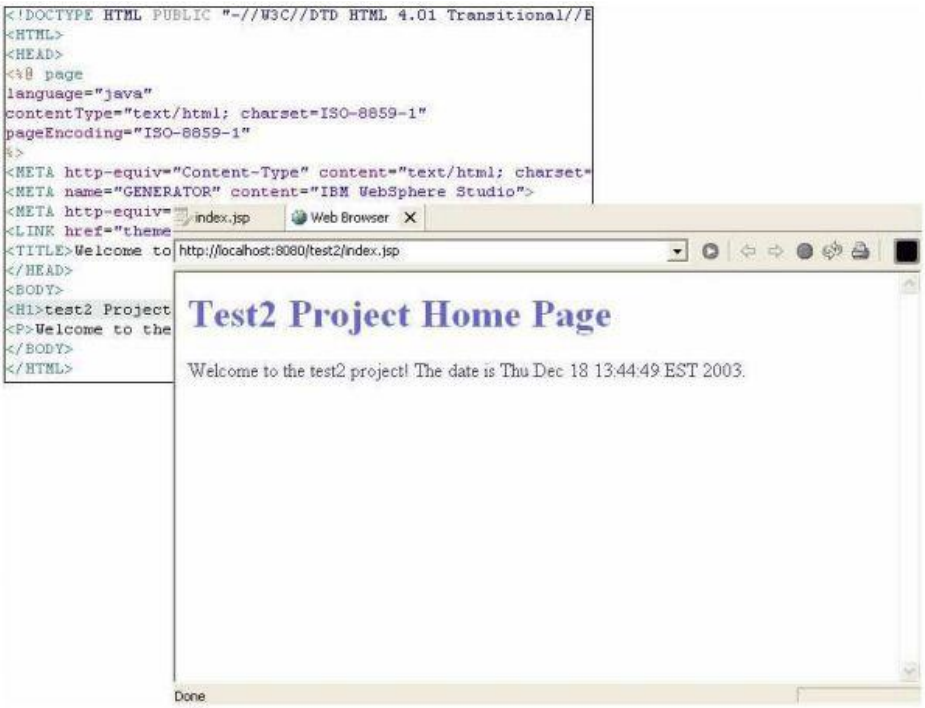
```
try {
Class.forName("com.mysql.jdbc.Driver").newInstance();
Connection con =
DriverManager.getConnection("jdbc:mysql://db.osu.cz/s
tag?user=java&password=heslo");
Statement stmt = con.createStatement();
}
catch (SQLException e) {}
```

PŘÍLOHA Č. 2 : PREZENTAČNÍ LOGIKA (JSP, JSF)

Java Server Pages - JSP (JSR 245 – JSP 2.1) umožňují dynamicky generovat HTML, XML či další typy dokumentů jako odpověď na požadavek (request) klienta, dále umožňují použít Java kód (znovupoužít EJB – Enterprise Java Beans) a předdefinované akce spolu se statickým obsahem. JSP syntaxe umožňuje také přidávat značky ve stylu XML pro volání vestavěné funkcionality.

JSP jsou jakousi nadstavbou nad servlety, umožňují lepší práce s HTML než servlety. Samotné JSP jsou však do formy servletů kompilovány. Od verze 2.0 je možné využívat také šablony pro generování pomocí výrazového jazyka (EL – expression language). Soubory JSP mají příponu .jsp.

Obrázek Pr2-1: Zdrojový kód JSP a jeho vzhled v prohlížeči.



JSP dovolují využívat speciální značky (tagy), které umožňují vkládat java výrazy, objekty, zkrátka java funkcionalitu do JSP stránek. Mezi základní tagy patří deklarace, výrazy, akce, direktivy. Některé z nich si nyní ukážeme.

Deklarace (declaration) slouží k deklaraci proměnných a metod v jazykce Java v JSP stránce, definujeme ji tagem*:* <%! … %>.

Příklad použití*:* <%! private int counter = 0 ; %>

Výrazy (expression) slouží pro vložení Java výrazů do JSP stránky, zapisujeme je značkou: <%= … %>.

Příklad použití: Date : <%= new java.util.Date() %>

Direktivy (directive) definují speciální informace o stránce, zda následuje vnoření stránky, zneplatnění session apod. Podle toho pak definujeme různé tagy: page, include, tag. K definici direktivy slouží tag: <%@ directive ... %>.

Akce (action) slouží pro použití JavaBeans, předání kontroly mezi stránkami nebo pro podporu appletů. Dále je možné vytvářet tzv. skriptlety (scriptlets).

Java definuje ještě další specifikace pro zobrazení dat. Jednou z nich je také **JSF** (Java Server Faces). Jedná se o webový aplikační framework, který je založený na komponentách, tedy ne na request/response modelu jako servlety, resp. JSP, i když vlastní JSF používají pro zobrazení JSP. Lze však použít jiných technologií zobrazení, například také XUL (prohlížeč Mozilla byl napsán v XUL) apod. JSF pracují tak, že před zavoláním dotazu (request) na novou stránku je stav komponenty uživatelského rozhraní uložen a obnoven po vrácení odpovědi od serveru. JSF specifikaci 2.0 nalezneme pod JSR 314.

PŘÍLOHA Č. 3 : ENTERPRISE JAVA BEANS (EJB)

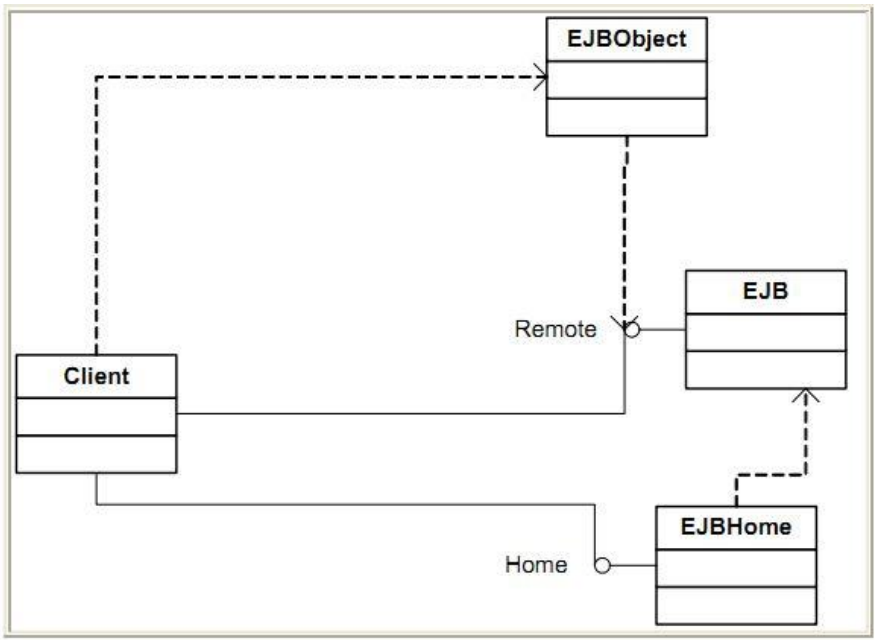
Nejdůležitejší a možná také nejsložitější specifikací Java EE jsou takzvané **EJB** (Enterprise Java Beans). Ty slouží k popisu logiky rozsáhlých aplikací. EJB standard je architektura pro serverové komponenty v Javě. Jedná se o jakýsi kontrakt mezi komponentou a aplikačním serverem. Aplikační server slouží jako prostředí pro spouštění EJB a nazývá se kontejner. Mezi hlavní výhody EJB patří:

- **Přenositelnost** – jelikož se jedná o standard a to nejen pro psaní EJB, ale také pro kontejner, je jednoduché vzít EJB a přenést jej do jiného kontejneru jiného výrobce.
- **Silná podpora kontejneru** – kontejnery již podle specifikace musí poskytovat velké množství služeb, které ulehčují samotný vývoj aplikace. Nicméně mnoho výrobců kontejnerů přidává ještě své nestandardní funkce.

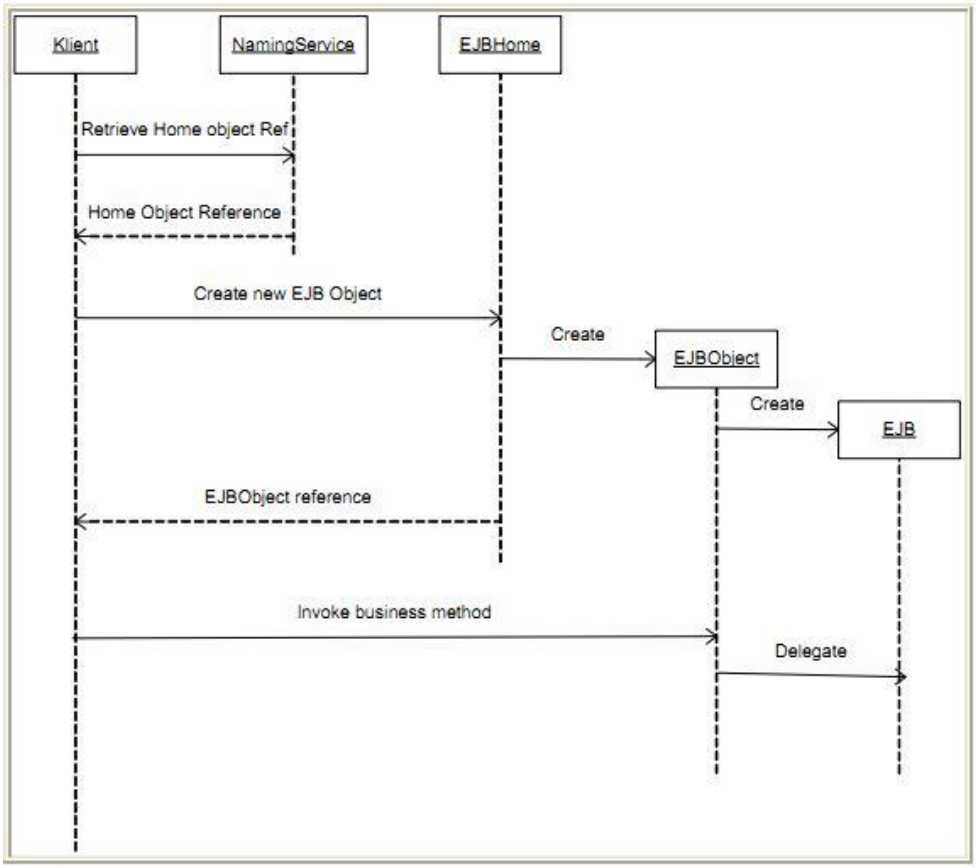
EJB komponenta se skládá z několika prvků, jak ukazuje obrázek níže. Tyto prvky jsou:

- **Bean** – jedná se o samotnou implementaci komponenty.
- **Remote Interface** – rozhraní definující služby, které bude EJB poskytovat.
- **EJB Object** – generovaná třída, která implementuje Remote Interface. Funguje jako proxy k EJB .
- **Home Interface** – rozhraní definující služby pro správu životního cyklu EJB.
- **Home Object** – jedná se o implementaci Home Interface. Funguje jako GoF Factory návrhový vzor. Stará se o životní cyklus EJB, jako je vytvoření či její destrukce.
- **Deployment deskriptor** – XML soubor definující vlastnosti EJB.

Obrázek Pr3-1: Základní prvky EJB.



Obrázek Pr3-2: Komunikace mezi elementy EJB.



Obrázek 1-6 popisuje základní schéma komunikace mezi klientem a EJB. Kroky jsou následující:

- Klient chce pracovat s vybranou EJB.
- Nalezne si její EJBHome objekt.
- Zavolá službu vytvoření a tak získá odkaz na EJBObject schovaný za rozhraní Remote.
- Klient může nyní využívat služeb EJB.
- Zavolá metodu definovanou v Remote.
- Volání se provede na EJBObject.
- Ten předá volání konkrétní implementaci.

Důvodem pro existenci EJBHome a EJBObject objektů je poskytnutí maximální flexibility a přidaných služeb. EJBHome může provádět tzv. pooling instancí EJB a tak limitovat maximální počet instancí a urychlit celkový proces inicializace. EJBObject zase zpřístupňuje služby kontejneru, jako je například práce s transakcemi, bezpečnost, či další. EJB definuje několik typů:

- Session
  - Stateless
  - Statefull
- Entity
  - Container managed persistency
  - Bean managed persistency
- Message driven bean

PŘÍLOHA Č. 4 : SERVLETY

Jednou ze specifikací Java EE jsou tzv. **servlety**. Jedná se o odpověď Javy na **CGI** (Common Gateway Interface) **skripty**. CGI skritpy měly několik nevýhod, mezi které patří nemožnost přístupu k adresářům serveru a tudíž dalším zdrojům, nemožnost přenosu skriptu na jiné platformy, či nutnost vždy spouštět danou instanci skriptu znovu. Servlety tyto problémy odstraňují:

- S každým dotazem se nespouští http (HyperText Transfer Protocol) požadavek, spouští se pouze thread (vlákno) Javy.
- Přenositelnost.
- Možnost přístupu na web server (pro obrázky, dokumenty, atd.).
- Existují volně dostupné (OpenSource), de facto standardní web servery (např. Apache).

Co to vlastně jsou ty servlety? Jedná se o API pro tvorbu dynamických webových aplikací. Je možné pomocí nich generovat kontext, jímž je často (X)HTML (HyperText Markup Language) nebo XML. Servlety běží na straně serveru a využívají modelu request/response[5]. Narozdíl od CGI scriptů umožňují **uchovávat stavové informace** (request/response model neumožňuje) a to pomocí cookies či sessions. Java Servlet API najdeme v balíčcích javax.servlet a javax.servlet.http. Výhodou servletů je existence speciálního běhového prostředí na straně serveru, kterému říkáme **kontainer**. Ten za nás řeší připojení (network connections), správu a dohodnutí protokolu (protocol negotiations), nahrávání tříd (class loading) a spoustu dalších důležitých věcí. Servlet tedy běží na webovém serveru (v rámci Servlet containeru) a je součástí stejného procesu jako tento web server, daný web server je tedy odpovědný také za inicializaci, volání a zrušení každé instance servletu. API Java Servlets nyní existuje ve verzi 2.5 a je součástí podnikové konfigurace Java EE 5.0, tuto verzi pak najdeme pod JSR 154. Servlety mají několik využití, mezi základní patří:

- Middle tier (spojení mezi klientem a back-end aplikací/službou) – to umožňuje použít „lehčí“ klienty, odlehčit zátěž serveru (balancování zatížení serverů podle potřeby) či connection/transaction management.
- Proxy vrstva v případě použití Java Appletů (např. e-banking, připojení k databázy).
- Podpora protokolů – všechny request/response orientované (SMTP, POP, FTP).
- Podpora HTML (dynamické generování zákaznických stránek, možnost využít gramatik, pravidel).

[5] Request-response anebo request-reply je jednou zezákladních metod, které používají počítače při vzájemné komunikaci. Použití request/response v praxi znamená, že když jeden počítač vyžaduje data, druhý počítač reaguje na požadavek odpovědí (jejich dodávkou).

PŘÍKLAD 125 : MYSQL PŘÍKLAD PŘIPOJENÍ

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWorld extends HttpServlet {
public void doGet(HttpServletRequest request,
HttpServletResponse response) throws
ServletException, IOException {
PrintWriter out = response.getWriter();
out.println("A Sample Servlet");
}
}
```

PŘÍLOHA Č. 5 : JAVA A FRAMEWORKY

Jazyk Java je podporován mnoha nástroji (komerčními, Open Source i volně dostupnými), existuje také spousta frameworků, které podporují vývoj Java aplikací (webových, distribuovaných, mobilních, atd.), jejich sestavení, převod objektových dat do relačních databází a další vlastnosti.

APACHE ANT

Prvním z těchto nástrojů, který zmíníme je **Apache Ant**, který slouží pro sestavení Java aplikace. Běžně probíhá sestavení aplikace pomocí příkazů operačního systému (např. v Unix se jedná o příkaz make). Tato varianta má (či obecně může mít) několik nevýhod:

- Složitá syntaxe.
- Běžné chyby/problémy s mezerami, tabelátory.
- Java programy jsou přenositelné -> možné problémy při sestavení programu na jiných OS.

Řešením, které zmíněné problémy odstraňuje je právě Apache Ant. Jedná se o aplikaci sloužící k sestavení složitějších programů v Javě. Ant je hojně podporován IDE (Integrated Development Environment) a dalšími nástroji (např. Eclipse, javadoc, apod.). K definici buildu používá XML syntaxi. Kvalitu tohoto programu naznačují také různá ocenění, která za dobu své existence posbíral. Jedná se například o:

- SD Magazine: Productivity Award 2002.
- JavaWorld: Most Useful Java Community-Developed Technology.

Apache Ant vznikl při vývoji kontejneru Tomcat pro ulehčení práce vývojářů se sestavováním aplikace. Jak již bylo řečeno, zpracovává Ant XML soubory. Každý z nich je defaultně pojmenovaný *build.xml* a obsahuje:

- 1 projekt,
- nejméně 1 cíl (target),
- několik úkolů (tasks) – každý zpracovává nějaká třída.

Díky této struktuře je možné Ant jednoduše rozšířit, k čemuž slouží tzv. tasky. Pro potřebné rozšíření přidáme task a nasměrujeme na třídu, která ho zpracovává. Zajímavé tasky, které Ant může zpracovávat (nutno mít v *ant_home/lib* .jar soubory) jsou například:

- Javac – základ, podmínky překladu, kompilátor,
- Junit – spouštění unit testů jako tasky,
- Xalan – XSLT procesor.

JUNIT

Dalším z důležitých nástrojů, o kterém si něco povíme je **JUnit** z rodiny nUnit sloužící pro unitové testování Java aplikací. Tento testovací framework je distribuován pod GPL licencí. Autory frameworku jsou Kent Beck a Erich Gamma. Framework vznikl za účelem podpory testy řízeného vývoje (Test Driven Development) v extrémním programování (XP). K čemu všemu tedy tento framework slouží a k čemu ho lze použít?

- Automatizované testování kódu vývojářů.
- Detailní dokumentace kódu/rozhraní (agilní přístup k dokumentaci).
- Zlepšení návrhu (nejdříve píšeme test, až potom vlastní implementaci, důsledkem je promyšlenější architektura).
- Lokalizace chyb (díky unit testu přesně víme, v které třídě či dokonce metodě se nachází chyba, snížíme čas potřebný na její lokalizaci).

Je možné definovat automatické spuštění unit testů pomocí JUnit frameworku jako Ant task při sestavení aplikace, což podporuje iterativní způsob vývoje s častým testováním a neustálou integrací vyvíjených částí (denní buildy), čili tzv. Continuous Integration (CI).

JAKARTA STRUTS

Jakarta Struts je aplikační framework (sada spolupracujících tříd a rozhraní pro řešení specifického problému, opakovaně použitelné) podporující a urychlující vývoj webově orientovaných aplikací. Jakarta Struts podporuje MVC (Model-View-Controler) model (architektura JSP Model 2), resp. aplikace je strukturována podle tohoto modelu. Struts obsahuje cca 250 tříd a rozhraní. Díky MVC modelu je oddělená aplikační logika, prezentační vrstvy a zpracování požadavku. Zpracování probíhá tak, že je požadavek uživatele zaslán řídícímu servletu (možnost centralizace činností), ten je po zpracování přesměrován na konkrétní JSP stránku.

Struts framework umožňuje využívat a podporuje všechny obvyklé prvky web aplikací, jako jsou Java servlety, JSP stránky a knihovny tříd, (X)HTML stránky, různá multimédia, textové dokumenty apod. Struts definuje také adresářovou strukturu aplikace, ta sestává z privátních adresářů, které jsou přes web nepřístupné, jedná se o:

- /WEB-INF – konfigurace aplikace,
- /WEB-INF/classes – kompilovaný kód aplikace,
- /WEB-INF/lib – pomocné knihovny aplikace.

Aplikace samotná je potom přístupná přes kořenový adresář. Ke konfiguraci aplikace slouží několik souborů, hlavními jsou web.xml, který slouží k mapování řídícího servletu a nastavení jeho parametrů a soubor struts-config.xml sloužící pro mapování jednotlivých akcí na Java třídy, datových zdrojů a výjimek.

HIBERNATE

Hibernate je framework pro objektově relační mapování (ORM) v Javě. Poskytuje funkčnosti, které nám umožní mapovat doménový objektový model do relační databáze (odděluje logický a fyzický datový model). Z našeho kódu tedy můžeme odstranit persistentní vrstvu, která se stará o transformaci a ukládání objektů do databáze. Hibernate je open source projekt (GNU Lesser GPL).

Mezi základní rysy tohoto frameworku patří to, že řeší tzv. Impedance mismatch, nesoulad mezi objektově orientovanými technologiemi a uložením jejich dat ve formě relací. Hibernate provádí mapování Java tříd do databázových tabulek (a mapování datových typů Javy do datových typů SQL). Lze použít se samostatnou Java aplikací stejně jako s Java EE aplikací (servlety, EJB session beany). Hibernate tedy vývojáře odstiňuje od ručního zpracování resultSetu a umožňuje přenositelnost na jakoukoli relační SQL databázi s pouze velmi malou režií. API Hibernate poskytuje balíček org.hibernate obsahující rozhraní:

- org.hibernate.SessionFactory – vytváří hibernate sessions (jako Singleton).
- org.hibernate.Session – reprezentuje hibernate session, správa stavu persistence, získání objektu z databáze a správa transakcí.