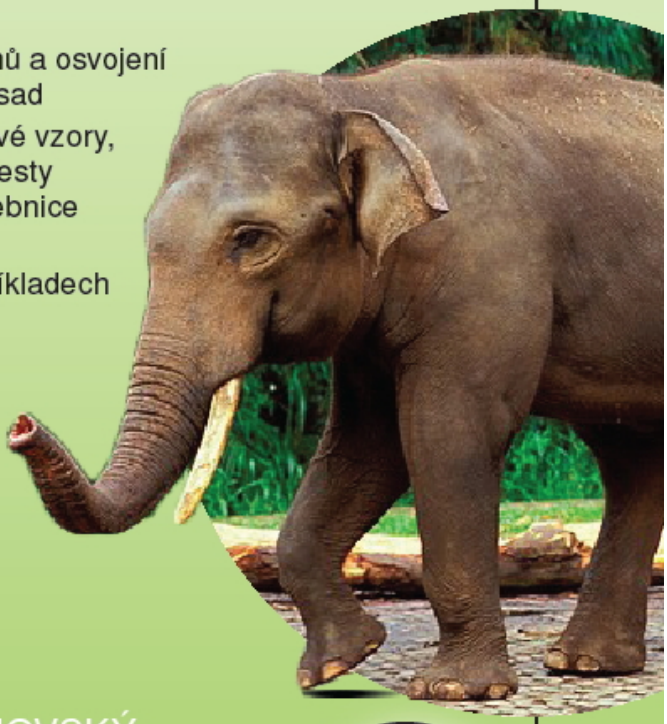
EDICE
myslíme v ...

knihovna programátora

- Učebnice pro ty, kteří nechtějí zůstat obyčejnými kodéry, ale chtějí se stát špičkovými architekty
- Probírá novinky Javy 8, které ovlivňují návrh architektury programu
- Soustředí se na návrh programů a osvojení klíčových architektonických zásad
- Vysvětluje a procvičuje návrhové vzory, refaktoraci kódu, vývoj řízený testy a další oblasti, které běžné učebnice ignorují
- Vše průběžně procvičuje na příkladech řešených spolu se čtenářem
- Doporučená učebnice na řadě středních škol i univerzit



RUDOLF PECINOVSKÝ

Java 8

Úvod do objektové architektury pro mírně pokročilé

GRADA

Java 8

úvod do objektové architektury pro mírně pokročilé

Rudolf Pecinovský
2014

Rudolf Pecinovský

Java 8

Úvod do objektové architektury pro mírně pokročilé

TIRÁŽ TIŠTĚNÉ PUBLIKACE

Vydala Grada Publishing a.s.
U Průhonu 22, Praha 7
jako svoji 5670. publikaci

Odborní lektoři:
doc. Ing. Pavel Herout, Ph.D.,
doc. MUDr. Jiří Kofránek, CSc.,
doc. Ing. Vojtěch Merunka, Ph.D.,
doc. Ing. Miroslav Virius, CSc.

Odpovědný redaktor: Martin Vondráček, Ladislava Soukupová
Návrh vnitřního layoutu: Rudolf Pecinovský
Zlom: Rudolf Pecinovský
Počet stran 656
První vydání, Praha 2014
Vytiskla tiskárna PROTISK, s. r. o.

ISBN 978-80-247-4638-8

TIRÁŽ ELEKTRONICKÉ PUBLIKACE

ISBN 978-80-247-9480-8 (ve formátu PDF)
ISBN 978-80-247-9481-5 (ve formátu EPUB)

Mé ženě Jarušce a dětem
Štěpánce, Pavlínce, Ivance a Michalovi

Stručný obsah

Podrobný obsah

Skrytí spoluautoři

Při tvorbě takto rozsáhlé knihy se autor nemůže spolehnout pouze sám na sebe. Je známou věcí, že když po sobě autor čte svůj rukopis, čte velmi často to, co chtěl napsat, a ne to, co doopravdy napsal. S kontrolou jazykových chyb mu může pomoci redaktor a jazykový korektor. S kontrolou odborných zaškobrtnutí mu však musí pomoci někdo jiný.

S kontrolou odborné správnosti musí pomoci někdo, kdo je odborně na výši a odhalí, že se autor vyjadřuje nepřesně nebo dokonce chybně. S kontrolou srozumitelnosti výkladu musí zase pomoci někdo, kdo se chce látku naučit a je ochoten text podrobně pročíst a upozorňovat na místa, kde výkladu zcela nerozumí nebo se mu zdá, že by bylo vhodné ještě něco doplnit. Úplně dokonalé pak je, když váš text čte zkušený učitel, který umí odhalit nejenom oba výše popsané druhy nepřesností či dokonce chyb, ale také naznačit možná potenciální nedorozumění a dezinterpretace vzniklá z podobnosti vykládané látky s jinými oblastmi, o nichž se výklad nezmiňuje.

Měl jsem to štěstí, že se mi podařilo získat spolupracovníky (a tím i kritiky) ze všech tří zmiňovaných skupin. Všem bych jim chtěl tímto poděkovat. Řekl bych, že díky jejich připomínkám a doporučením je výsledný text kvalitnější.

Všechny tyto dobrovolné spolupracovníky vnímám jako spoluautory. Proto mi dovolte, abych je tu alespoň abecedně vyjmenoval – byli to: Milan Augustin, Tibor Bako, Jakub Hadam, Michael Charvát, Jiří Kofránek, Jiří Kubala, Petr Kuchař, David Král, Filip Malý, Nikolas Patrik, Jarmila Pavlíčková, Luboš Pavlíček, Josef Svoboda, Petr Vidrman, Martin Vondráček a Martin Žamberský.

Úvod

Otevíráte knížku, která vás chce naučit programovat moderním, objektově orientovaným stylem. Stylem, jímž se v dnešní době vyvíjí drtivá většina klíčových aplikací, ale k jehož výuce ještě řada škol nedospěla. Po nastudování této knížky budou proto mnozí z vás vědět o moderním programování víc než leckterý z vašich učitelů.

Komu je kniha určena

Kniha je určena programátorům, kteří potřebují získat hlubší vhled do problematiky objektově orientovaného programování a návrhu architektury objektově orientovaných programů. K jejímu studiu přitom nepotřebují žádné velké předběžné zkušenosti. Stačí znalosti na úrovni mých začátečnických učebnic.

Kniha je reakcí na nepřetržité nářky vedoucích programátorských týmů, kteří posílají nastoupivší absolventy do mých přeškolovacích kurzů. Stěžují si, že školy opouštějí možná skvělí kodéři, ale neschopní architekti, že tito absolventi znají několik programovacích jazyků a řadu užitečných frameworků, ale mají problém s návrhem kvalitní architektury zadávaného programu.

Většina škol, učebnic a výukových kurzů se soustředí především na výklad syntaxe probíraného programovacího jazyka a probrání hlavních knihoven používané platformy. Jejich autoři se soustřeďují na detaily v bláhové naději, že jejich studenti a čtenáři pak v průběhu následující praxe pochopí vyšší principy. Když se ve výukovém programu objeví kurzy návrhových vzorů, končí často jako defilé technik, jak to či ono šikovně zakódovat.

Koncepce knihy

Ve svých knihách se pokouším tento stereotyp změnit. Soustředím se především na předvádění toho, jak program navrhnout a probrat věci, které považuji za důležité a přitom je v jiných učebnicích většinou nenajdete. Přitom se pokouším vše demonstrovat na příkladech. Na rozdíl od některých učebnic se však nechci omezovat pouze na AHA-příklady, jejichž jediným cílem je, aby po jejich prostudování student prohlásil „Aha! Takto to funguje!“, i když se jim v některých chvílích nevyhnu. Snažím se ale co největší část látky demonstrovat na složitějších příkladech, které však nejsou zahlceny spoustou šumu, za který považuji kód, který se

musí naprogramovat, i když s vykládanou látkou přímo nesouvisí. Dávám přitom přednost příkladům používajícím grafiku, protože jsou pro daný účel většinou nejnázornější.

Kniha je koncipována jako druhý díl knihy *Java 7 – Učebnice objektové architektury pro začátečníky*[1]. Předpokládá proto, že znáte látku zhruba na úrovni prvního dílu. Protože ale očekávám, že knihu bude číst i řada čtenářů, kteří získali své první programátorské zkušenosti z jiných zdrojů, tak pro jistotu na počátku nejdůležitější věci velice stručně zopakuji.

Co se naučíte, uspořádání knihy

Snažil jsem se v knize soustředit na oblasti, které se do jiných knih nevešly, a přitom je jejich osvojení velmi důležité, protože jejich neznalost je v lepším případě příčinou těžkopádných řešení a v horším případě příčinou špatně odhalitelných chyb.

V první části nejprve bleskově zopakuji základní látku prvního dílu a pak vás seznámím s jedním ze tří nejpoužívanějších profesionálních vývojových prostředí: s prostředím *NetBeans*. V minulém dílu jsme pracovali s vývojovým prostředím *BlueJ*. To je sice asi nejlepší prostředí pro získání základních představ a návyků objektově orientovaného programování, ale pro profesionální práci se příliš nehodí. Protože je ale zvládnutí vývojového prostředí zhruba stejně náročné jako zvládnutí programovacího jazyka, je mu věnován celý zbytek první části. Naučíte se je konfigurovat, vytvářet v něm projekty a využívat jeho výhodných vlastností pro zefektivnění své práce.

V druhé části si postupně ukážeme různé programové konstrukce a postupy, které můžete využít při návrhu architektury projektu. Postupně se seznámíte s dalšími návrhovými vzory, osvojíte si definici interfejsů obsahujících definice metod včetně jejich implementace, dozvíte se o možnosti parametrizace datových typů a jejich vzájemném vnořování, naučíte se používat lambda-výrazy, které umožňují pracovat s částmi kódu jako s proměnnými, a získáte první zkušenosti s prací s kontejnery a datovody.

Ve třetí části se ponoříme do výkladu dědění implementace. Začneme podrobným výkladem zavádění objektů a tříd a budeme pokračovat základními vlastnostmi dědění implementace. Dozvíte se, jak vytvářet třídy, jejichž předky jsou jiné třídy, a hlavně na co si dát při konstrukci takovýchto tříd pozor. Pokusím

[1] Kniha je současně slibovaným druhým dílem starší učebnice *OOP – Naučte se myslet a programovat objektově*. Ta sice vyšla v jiném nakladatelství, ale protože dané nakladatelství nedokázalo před několikeré upomínky dostát svým závazkům, tak jsem z tvorby druhého dílu vycouval.

se vám demonstrovat některé vlastnosti dědění na třídách, které dědění emulují využitím návrhového vzoru *Dekorátor* a předvedu vám, jak definovat společného rodiče skupině tříd, které mají některé společné vlastnosti.

Ve čtvrté, závěrečné **části**, vám představím algoritmické konstrukce, jejichž používání je sice postupně nahrazováno jinými technikami (ty probírají předchozí části učebnice), ale prozatím se bez nich stále ještě neobejdeme. Navíc se ve starších programech s některými novějšími konstrukcemi ani nesetkáte.

Programovací jazyk

Přestože má kniha ve svém názvu uveden jazyk Java, tak musím dopředu oznámit, že se nejedná o učebnici jazyka Java. Těch je k dispozici více než dost. Jedná se o učebnici objektově orientovaného programování a jazyk Java je zde používán jako jazyk, v němž jsou zapsány programy demonstrující probíranou látku. Tento jazyk jsem zvolil z několika důvodů:

- Je to stále s odstupem nejpoužívanější objektově orientovaný jazyk. Programátoři v Javě jsou stále nejžádanější (a mají i jedny z nejvyšších platů).
- Je to jazyk poskytující všechny klíčové konstrukce používané v moderním programování.
- Vytvořené programy nejsou omezeny na jediný operační systém, ale můžete je přenášet mezi různými operačními systémy.
- A vlastnost neocenitelná pro studium a začátečnické experimenty: nástroje pro vývoj všech druhů aplikací od čipových karet až po rozsáhlé aplikace běžící na několika počítačích můžete sehnat zdarma.

Potřebné vybavení

Pro úspěšné studium této knihy budete potřebovat:

- chuť naučit se objektově programovat a výdrž v situacích, kdy se vám nebude zcela dařit,
- rozumně výkonný počítač,
- základní vývojovou sadu Javy (JDK) ve verzi 8 a vyšší, kterou si můžete stáhnout se stránek http://www.oracle.com/technetwork/java/javase/downloads.
- vývojové prostředí *NetBeans* ve verzi 8 a vyšší, které si můžete stáhnout ze stránek https://netbeans.org.
- doprovodné příklady, které si můžete stáhnout ze stránky knihy http://knihy.pecinovsky.cz/uoa2_j8.

Doprovodné projekty

Na stránce knihy na adrese http://knihy.pecinovsky.cz/uoa2_j8 najdete také soubor s generátorem projektů použitých v knize. Téměř každá kapitola má svůj doprovodný projekt. Některé kapitoly jich mají dokonce několik. Možná bude někomu připadat zbytečné vytvářet takové množství projektů, když jsou jednotlivé sekce umísťovány do samostatných balíčků a řada projektů ve svém obsahu vychází z projektů předchozích, k nimž buď něco přidá a/nebo něco jiného drobně upraví. Důvodem tohoto uspořádání je snaha vyjít začátečníkům vstříc a umožnit jim v každé kapitole začít znovu s funkčním projektem nezávisle na tom, do jakého stavu přivedli projekt z kapitoly přechozí.

První díl byl určen pro naprosté začátečníky, kterým jsem se snažil vyjít vstříc, a protože vím, jaký problém s angličtinou má značná část studentů, používal jsem pro větší názornost české identifikátory. Pořízením této učebnice jste se přihlásili mezi pokročilejší programátory, a ti se bez znalosti angličtiny neobejdou. V této branži platí: *programátor musí umět anglicky, anebo musí změnit zaměstnání*. V tomto dílu jsou proto už všechny identifikátory anglicky. České zůstávají pouze dokumentační komentáře.

Doplňková literatura

Kniha je tlustá, ale ani tak se do ní nevešlo vše, co by bylo potřeba. Navíc se mi nechtělo podrobně rozebírat témata, která už jsem vysvětlil v publikaci, kterou si můžete zdarma stáhnout. Hovořím konkrétně o knize *Java 5.0 – Novinky jazyka a upgrade aplikací*, kterou si můžete zdarma stáhnout na mých webových stránkách na adrese http://knihy.pecinovsky.cz/java5novinky.

Kromě toho se na svých stránkách http://vyuka.pecinovsky.cz chystám postupně zveřejňovat materiály, které budu připravovat pro studenty a případné polotovary budoucích publikací. Když si uvědomím, že při psaní knih vychází honorář asi tak na 7 korun za hodinu práce, tak se domnívám, že nic neztratím, když budu budoucí texty poskytovat zdarma a ponechám na rozhodnutí čtenářů, zda si jich cení tak, že jsou za ně ochotni něco zaplatit.

Použité konvence

Místní nabídka

„Skrytí spoluautoři“, kteří četli vytvářený rukopis, mne upozornili na některé formulace, které je nebo jejich kolegy trochu zaskočily. Jednou z nich je termín *místní nabídka*, který se sice v počítačové literatuře používá od počátku devadesátých let, nicméně do programátorského slangu ještě nepronikl.

Opatrně proto připomenu, že budu-li někde hovořit o tom, že máte zadat nějaký příkaz v místní nabídce objektu `Xyz`, znamená to, že máte na tento objekt najet myší, stisknutím pravého tlačítka rozbalit jeho místní nabídku (anglicky context menu) a v ní zadat požadovaný příkaz.

Formátování

K tomu, abyste se v textu lépe vyznali a také abyste si vykládanou látku lépe zapamatovali, používám několik prostředků pro odlišení a zvýraznění textu.

Objekty První výskyt nějakého termínu a další texty, které chci zvýraznit, vysazuji **tučně**.

Názvy Názvy firem a jejich produktů vysazuji *kurzivou*. Kurzivou vysazuji také názvy kapitol, podkapitol a oddílů, na které se v textu odkazuji.

Citace Texty, které si můžete přečíst na displeji, např. názvy polí v dialogových oknech či názvy příkazů v nabídkách, vysazuji **tučným bezpatkovým písmem**.

Adresy Názvy souborů a internetové adresy vysazuji obyčejným bezpatkovým písmem.

`Program` Texty programů a jejich částí vysazuji `neproporcionálním písmem`.

Kromě částí textu, které považuji za důležité zvýraznit nebo alespoň odlišit od okolního textu, najdete v textu ještě řadu doplňujících poznámek a vysvětlivek. Všechny budou v jednotném rámečku, který bude označen ikonou charakterizující druh informace, kterou vám chce poznámka či vysvětlivka předat.

Symbol jin-jang bude uvozovat poznámky, s nimiž se setkáte na počátku každé kapitoly. Zde vám vždy prozradím, co se v dané kapitole naučíte.

Otevřená schránka s dopisy označuje informace o projektu, s nímž budeme v dalším textu pracovat, nebo v něm najdete vzorové řešení.

Obrázek knihy označuje poznámku týkající se používané terminologie. Tato poznámka většinou upozorňuje na další používané termíny označující stejnou skutečnost nebo na konvence, které se k probírané problematice vztahují. Seznam všech terminologických poznámek najdete v rejstříku pod heslem „terminologie“.

Obrázek počítače označuje zadání úkolu, který máte samostatně vypracovat. Seznam všech úloh najdete v rejstříku pod heslem „úloha“.

Píšící ruka označuje obyčejnou poznámku, která pouze doplňuje informace z hlavního proudu výkladu o nějakou zajímavost.

Ruka s hrozícím prstem upozorňuje na věci, které byste měli určitě vědět a na které byste si měli dát pozor, protože jejich zanedbání vás většinou dostane do problémů.

Usměváček vás bude upozorňovat na různé tipy, kterými můžete vylepšit svůj program nebo zefektivnit svoji práci.

Mračoun vás naopak bude upozorňovat na různá úskalí programovacího jazyka nebo programů, s nimiž budeme pracovat, a bude vám radit, jak se těmto nástrahám vyhnout či jak to zařídit, aby vám alespoň pokud možno nevadily.

Brýle označují tzv. „poznámky pro šťouraly“, v nichž se vás snažím seznámit s některými zajímavými vlastnostmi probírané konstrukce nebo upozorňuji na některé souvislosti, jejichž znalost však není k pochopení látky nezbytná.

Odbočka

Občas je potřeba vysvětlit něco, co nezapadá přímo do okolního textu. V takových případech používám podšeděný blok se silnou čarou po straně. Tento podšeděný blok je takovou drobnou odbočkou od ostatního výkladu. Nadpis podšeděného bloku pak najdete i v podrobném obsahu mezi nečíslovanými nadpisy.

Část I:
Vývojové prostředí

Tato část vám nejprve připomene,
co byste měli znát z prvního dílu,
abyste mohli bez problémů pokračovat ve čtení knihy.
Pak vás postupně seznámí s vývojovým prostředím *NetBeans*
a jeho základními vlastnostmi,
které by vám mohly pomoci výrazně zefektivnit vývoj programů.

Kapitola 1
Co byste měli znát z prvního dílu

Co se v kapitole naučíte
V této kapitole si velmi stručně připomeneme, co jste se naučili v prvním dílu učebnice, a seznámím vás s několika termíny, které budeme v dalším textu používat.

1.1 Přehled látky prvního dílu

V první verzi rukopisu této knihy jsem z minulého dílu převzal podkapitoly nazvané *Shrnutí – co jsme se naučili*, v nichž jsem na konci každé kapitoly vždy stručně vyjmenoval, co se v dané lekci probíralo. Přitom jsem s údivem zjistil, že tento velmi zhuštěný přehled má přes 30 stránek. To mne zaskočilo. I když jsem jej prošel a vyjmul vše, co se týkalo jen *BlueJ* a projektů vytvářených v prvním dílu, stále mi ještě připadal na úvodní kapitolu příliš velký.

Rozhodl jsem se proto, že původní text úvodní kapitoly vyjmu z učebnice, nepatrně jej upravím a vystavím v samostatném souboru na webových stránkách knihy, odkud si jej budou moci stáhnout a přečíst i ti, kteří první díl nečetli a rozhodli se začít rovnou tímto druhým dílem (zmíněný text najdete na stránce knihy). V dalším textu budu předpokládat, že vše, co je v onom stručném shrnutí uvedeno, již znáte a mohu na to plynule navázat.

Pro jistotu ale ještě v následujících pasážích připomenu některé klíčové termíny, které jsem v prvním dílu zavedl a které nemusí být zcela srozumitelné těm, kteří získávali prvotní informace z jiných zdrojů než z prvního dílu. Zkušenost totiž ukazuje, že řada studentů tyto termíny zaměňuje, směšuje a/nebo je špatně

interpretuje. Stručný přehled klíčových termínů uvádím především proto, abyste v okamžiku, kdy narazíte na daný termín v textu, věděli, o čem hovořím.

1.2 Definice × deklarace

Než začneme s vlastním opakováním, chtěl bych připomenout dva obecné jazykové termíny, které si studenti občas pletou, a to termíny *deklarace* a *definice*:

- **Deklarace** je část kódu, v níž předem oznamujeme některé vlastnosti vytvářeného programu. Je to takové prohlášení. Účelem deklarace je poskytnout překladači informace, na jejichž základě pak může kontrolovat, jestli ve svém programu neděláte nějaké evidentní chyby.
- **Definice** je část kódu, v níž něco doopravdy vytváříme. Jedná-li se o metodu, opravdu se někde objeví předpis definující, jak metoda pracuje. Jedná-li se o objekt, necháváme mu přidělit paměť a přiřadit počáteční hodnotu.

U datových objektů (proměnných a konstant) se často slučuje deklarace s definicí. Danou proměnnou deklarujeme (prozradíme, jaký bude mít datový typ a jak se bude jmenovat) a současně ji i definujeme (přiřadíme jí počáteční hodnotu, čímž donutíme překladač, aby pro ni vyhradil potřebnou paměť).

Klasické abstraktní metody, s nimiž se setkáme v interfejsech, jsou zde pouze deklarovány. V deklaraci pouze překladači vysvětlujeme, jakou by měla mít požadovaná metoda signaturu, jaká by měla být, kdyby ji někdo definoval.

Ve třídách jsme pak metody přímo definovali, protože jsme za hlavičkou metody zapisovali její tělo, v němž jsme popisovali, co má daná metoda dělat.

V tomto dílu se dozvíte, že situace není tak jednoduchá, jak se původně zdálo, tj. že se v interfejsech metody deklarovaly a ve třídách definovaly. Dozvíte se, že ve třídě můžete deklarovat abstraktní metodu a naopak v interfejsu definovat konkrétní metodu. K tomu všemu postupně dojdeme.

1.3 Co je to objekt

Vraťme se ale k objektovému programování. Říkali jsme si, že každý program je simulací reálného nebo virtuálního světa. Tento svět je tvořen objekty, které jsou v programu reprezentovány pojmenovanými daty. Jako **objekt** označujeme v programu kolekci dat reprezentující odpovídající objekt z onoho simulovaného světa.

Oproti běžnému chápání objektů v reálném světě je však v programu používán termín objekt mnohem obecněji: objekty v programu mohou reprezentovat cokoliv, co můžeme pojmenovat, tj. cokoliv, co můžeme označit podstatným jménem.

V programu proto chápeme jako objekty i vlastnosti (velikost, směr, krása), děje (připojení, přerušení, výpočet) a další pojmenovatelné entity. Objektem je proto i třída, rozhraní, metoda atd.

Malá poznámka pro ty, kteří mají zkušenosti s jazyky *Turbo Pacal* a *Delphi*. Jejich autor nám přichystal malý terminologický guláš, protože to, co zbytek světa označuje termínem *třída*, začal nazývat *objekt*. Možná proto budete chvíli zmateni, ale věřím, že jenom chvíli.

1.4 Datový typ, třída, class-objekt

V programech se vyskytují celé skupiny objektů, které mají stejnou charakteristiku. Tuto charakteristiku označujeme jako *datový typ* oněch objektů. Objekty s charakteristikou definovanou nějakým datovým typem označujeme jako *instance daného datového typu*.

Datové typy lze rozdělit do několika skupin. Jazyk Java rozlišuje následující druhy datových typů:

- *Primitivní datové typy* představují typy, které nejsou objektové (nemají své instance), ale zato mají přímou podporu v instrukčním souboru většiny procesorů. Nabourávají čistou objektovou orientovanost jazyka a byly do něj zavedeny pro zvýšení efektivity. Prozatím se bez nich neobejdeme, ale to by se mělo v některé z příštích verzí změnit.
- *Třídy* představují základní druh objektového datového typu.
- *Výčtové typy* jsou speciální třídy, které mají pouze předem definované instance, přičemž tuto množinu instancí není možno dále měnit.
- Interfejsy představují datové typy reprezentující rozhraní instancí tříd, které tyto interfejsy implementují (podrobněji za chvíli).

Datové typy jsou zvláštním druhem objektů. V jazyce Java nejsou (na rozdíl např. od jazyka Smalltalk) instancí žádného datového typu. Můžeme je však oslovovat prostřednictvím speciálních zástupných objektů, které jsou instancemi třídy `java.lang.Class` (podrobněji se s nimi seznámíme v pasáži *Třídy jako objekty – class-objekt třídy* na straně 123).

1.5 Zpráva × metoda, polymorfismus

Objekty simulovaného světa spolu navzájem komunikují – fyzikálně, chemicky, biologicky, verbálně, společensky atd. Veškeré tyto interakce reprezentujeme v programu jako zasílání zpráv jednoho objektu jinému.

Zpráva Termínem *zpráva* označujeme požadavek, který zasílá jeden objekt druhému. Odesílatel zprávy přitom očekává od adresáta reakci odpovídající kontraktu dané zprávy (co je to kontrakt, si připomeneme za chvíli).

Metoda Termínem *metoda* označujeme kód definující, jak bude daný objekt reagovat na zaslanou zprávu.

Každá zpráva, na kterou umí objekt zareagovat, má přiřazenu metodu, v níž je definována požadovaná reakce objektu na zaslání dané zprávy. Některým zprávám ale může být přiřazeno více metod a to, která metoda se použije, záleží na různých věcech. Většina jazyků (a Java mezi nimi) odvozuje to, jaká metoda bude použita, od typu osloveného objektu; obecně to ale může záviset i na jiných věcech.

Schopnost objektů reagovat na stejnou zprávu různě označujeme jako **polymorfismus**. Takové chování ale pouze reprezentuje stav v onom světě simulovaném naším programem. Představte si např. nějaký sportovní zápas. Všichni v hledišti jsou instance typu `Člověk`. Nicméně na zprávu o vstřeleném gólu budou instance podtypu `FanoušekTýmuA` reagovat jinak než instance podtypu `FanoušekTýmuB` a opět jinak budou reagovat instance podtypu `ProdavačkaNaTribuně`.

1.6 Rozhraní × interfejs

Pro ty, kteří nečetli minulý díl, bych připomněl, že v textu učebnice rozlišuji mezi termíny rozhraní a interfejs:

- **Rozhraní** představuje množinu charakteristik, jimiž se daná entita (balíček, třída, metoda, atribut, …) prezentuje svému okolí, a které tím pádem okolí o dané entitě explicitně zná. Své rozhraní má každá entita programu.
- **Interfejs** je programová konstrukce, kterou Java zavedla a kterou od ní mnohé další programovací jazyky převzaly. Díky interfejsu můžeme specifikovat požadavky na rozhraní objektů, s nimiž se chystáme pracovat.

Některé programátory irituje, že používám počeštěnou podobu anglického *interface*. V tištěném textu však musíme dávat přednost spisovným tvarům před slangovými. Převzaté slovo interface nelze skloňovat, a proto dávám přednost povolenému (viz *Akademický slovník cizích slov*. Academia Praha 1995. ISBN 80-200-0497-1) počeštěnému tvaru. Ostatně počeštění tvaru často používaných převzatých slov je běžné – podívejte se např. na slova fotbal, hokej či tenis.

V minulém dílu jsem navíc využil toho, že interfejs sloužil opravdu jenom k popisu požadovaného rozhraní, a pokud jsem se domníval, že nemůže dojít k nedorozumění, používal jsem český termín rozhraní i při označování dané programové konstrukce. To už si v tomto dílu nemohu dovolit, protože v Javě 8 přestal interfejs pouze deklarovat požadované rozhraní. Java 8 totiž umožňuje, abychom v interfejsu definovali i doporučenou implementaci (viz kapitolu *14 Implicitní implementace – `RingVehicle`, `ControlledVehicle`* na straně 300), a navíc může obsahovat i definice statických metod. Protože se nyní významy obou termínů vzdálily, budu jejich terminologické rozdělení důsledně dodržovat.

V souvislosti s rozhraním bych ještě připomněl další dva termíny: signatura a kontrakt:

- **Signatura** označuje tu část rozhraní, kterou může zkontrolovat překladač. Sem patří např. název dané entity, její datový typ (případně typ metod návratové hodnoty), u metod typy jednotlivých parametrů a některé další překladačem kontrolovatelné charakteristiky.
- **Kontrakt** označuje tu část rozhraní, kterou překladač zkontrolovat nedokáže – např. že šířka obdélníku nesmí být záporná. Kontrakt popisujeme v dokumentačních komentářích a za jeho dodržení je zodpovědný programátor. Za prvé tak, že sám kontrakty dodržuje, a za druhé tak, že za běhu programu kontroluje, že je dodržují i používané objekty (s jednou ověřovací konstrukcí se seznámíme v podkapitole *9.15 Ověřování podmínek – příkaz `assert`* na straně 222).

1.7 Zapouzdření a skrývání implementace

Vedle rozhraní má každá entita programu svoji implementaci, která specifikuje, jak daná entita dosahuje funkčnosti deklarované v jejím rozhraní. Jedna z hlavních zásad správného programování doporučuje programovat tak, abyste v každé situaci využívali pouze znalostí rozhraní používané entity a nesnažili se vylepšit program díky znalostem její implementace.

Zásada se nejlépe dodržuje tak, že nemám šanci ji porušit. Proto moderní programovací jazyky umožňují maximálně skrývat implementační detaily.

V souvislosti se skrýváním implementace se často hovoří o zapouzdření. Tímto termínem byla původně označována skutečnost, že objektové programování zapouzdřuje do společného objektu data i kód, který s těmito daty pracuje. Postupem času původní význam termínu *zapouzdření* ustupoval do pozadí a stále častěji se tímto termínem označovalo především skrývání implementace. Podvolím se tomuto všeobecnému trendu a budu jej v tomto významu používat i v této učebnici.

1.8 Datové typy a jejich dědění

V minulém dílu jsme se seznámili s konstrukcí dědění datových typů. Vysvětlili jsme si, že datový typ potomka specifikuje společné vlastnosti nějaké speciální podmnožiny instancí předka, přičemž objekty z této podmnožiny mají oproti ostatním instancím předka jisté speciální vlastnosti, kvůli nimž je pro ně vhodné definovat samostatný datový typ. Současně jsem vám prozradil, že existují tři typy dědění:

- **Přirozené (nativní) dědění** hovoří o tom, jak specializaci objektů cítíme bez ohledu na to, jak ji naprogramujeme.

  Ve škole jsme se např. učili, že čtverec je speciální případ obdélníku. Vysvětlovali jsme si, kdy můžeme tuto obecnou znalost promítnout do našeho programu, a kdy si to naopak dovolit nemůžeme.

- **Dědění typu** odpovídá v programech **dědění rozhraní**. Toto dědění se uplatňuje nejen při dědění interfejsů, ale i při implementaci interfejsu třídou. Z hlediska dědění typů lze proto implementovaný interfejs považovat za jednoho z rodičů tříd, které jej implementují.

  Připomínám, že **dědění typů je o slibech**. Když datový typ prohlásí, že dědí nějaký jiný typ, tak překladač může zkontrolovat pouze to, jestli dodržuje požadovanou signaturu. Na zodpovědnosti programátora pak je, aby vedle signatury dodržel i kontrakt, a splnil tak slib, který dal, když přihlašoval datový typ k dědění jeho rodiče. Nedodržení tohoto slibu je příčinou celé řady chyb.

- **Dědění implementace** se uplatní při dědění tříd. Při něm třída od svého rodiče přebírá nejenom jeho rozhraní, ale překladač zabezpečí, že přebere také veškerou jeho implementaci. Nemusí tedy definovat pro všechno vlastní metody, ale v řadě případů může použít metodu zděděnou od svého rodiče či od některého z prarodičů.

V dědění implementace jsou skryty mnohé záludnosti (na některé z nich jsem v minulém dílu upozorňoval), a proto jsem jeho výklad odložil až do tohoto dílu. Budeme o něm hovořit ve třetí části, kdy budete mít větší zkušenosti, které vám za prvé pomohou:

- odhalit situace, v nichž je použití dědění implementace nešikovné či přímo nevhodné (velmi oblíbená chyba, kterou dělají mnozí zdánlivě zkušení programátoři),
- orientovat se v situacích, kdy je použití dědění implementace výhodné, ale hrozí při něm riziko zanesení „oblíbených" chyb.

Má-li být náš program stabilní a v budoucnu snadno rozšiřitelný, musíme při dědění vždy respektovat všechny tři aspekty.

Vlastní instance třídy a mateřská třída objektu

Abych usnadnil orientaci v příslušnosti objektu k datovým typům, zavedl jsem ještě další dva termíny:

- **Vlastní instance třídy** (anglicky *class' own instance*) je instance, kterou „porodila" daná třída, a není proto instancí žádného z jejích potomků.
- **Mateřská třída objektu** (anglicky *home class*). Je-li objekt vlastní instancí nějaké třídy, označujeme tuto třídu jako *mateřskou třídu* daného objektu.

LSP – Liskov Substitution Principle

Před chvílí jsem říkal, že odvozený typ specifikuje společné vlastnosti podmnožiny instancí předka. Z vlastnosti, že prvky podmnožiny nepřestávají být prvky původní množiny, vyplývá, že instance potomků nepřestávají být instancemi předka, a měly by proto být schopny kdykoliv vystupovat v roli instance kteréhokoliv ze svých předků.

Zásadu, že **potomek musí být schopen kdykoliv plnohodnotně vystupovat v roli instance předka**, formulovala v roce 1987 ve své přednášce[2] Barbora Liskov a od té doby je označována jako *Liskov Substitution Principle* (*Substituční princip Liskové*) a používá se pro ni zkratka LSP.

---

[2] Liskov, B. 1987. Keynote address – data abstraction and hierarchy. In *Addendum To the Proceedings on Object-Oriented Programming Systems, Languages and Applications (Addendum)* (Orlando, Florida, United States, October 04 – 08, 1987). L. Power and Z. Weiss, Eds. OOPSLA '87. ACM, New York, NY, 17-34. DOI= http://doi.acm.org/10.1145/62138.62141

Bohužel, řada programátorů tuto zásadu ve svých programech porušuje, a pak se diví, proč jim jejich „dokonalé programy“ nechodí. S příklady porušujícími tuto zásadu se bohužel setkáte i v přednáškách řady universit (včetně těch nejprestižnějších) o učebnicích a kurzech programování ani nemluvě.

Přetěžování × přebíjení × zakrývání metod

V souvislosti s děděním bych připomněl další tři termíny, které si začátečníci občas pletou, a to přetěžování, přebíjení a zakrývání metod.

- O **přetěžování** (anglicky *overloading*) metod hovoříme tehdy, definuje-li jeden datový typ několik metod se stejným názvem, ale rozdílnou signaturou.
- O **přebíjení** (anglicky *overriding*) nebo **překrývání** metod hovoříme tehdy, definuje-li potomek metodu se stejnou signaturou, jako má některá z viditelných metod rodiče. Potomkova verze metody je přitom začleněna do definice objektu tak, aby se použila pokaždé, když potomkovi někdo pošle příslušnou zprávu, a to nezávisle na tom, za instanci jakého typu se v danou chvíli potomek vydává. Přebíjení se budeme podrobněji věnovat v kapitole *24 Virtuální metody a jejich přebíjení* na straně 515.
- O **zakrývání** (anglicky *hiding*) metod hovoříme tehdy, pokud potomek definuje metodu se stejnou signaturou, jako má některá z metod jeho rodiče, avšak rodičovská metoda zůstává dostupná a to, která z metod se použije, záleží na tom, komu se daná zpráva posílá. V Javě je možno zakrývat pouze statické metody. Instanční metody je možné pouze přetížit či přebít. Zakrývání se budeme podrobněji věnovat v kapitole *23 Zakrývání atributů a metod* na straně 498.

Překlad *přebít* přesněji odpovídá anglickému originálu, a proto mu budu dávat přednost. V minulém dílu jsem si to ještě netroufl a dal na námitky kolegů, že takový termín je příliš lidový a do odborné terminologie nepatří. Pak jsem si ale řekl, že nemusíme být tak úzkoprsí. Když mohou být lidové výrazy v angličtině, proč bychom je nemohli mít v češtině. Angličan či Američan klidně označí programový modul termínem *podniková fazole* (*enterprise bean*). Našim programátorům z takového označení naskakuje kopřivka. Pokud máte s takovýmito termíny mentální problémy, doporučuji vám počkat si na vydání anglické verze této učebnice. Tam budu sice používat tytéž výrazy, ale v anglickém textu takovéto termíny nikoho neuráží.

1.9 Odkazové a hodnotové datové typy

Objektové datové typy rozdělujeme na odkazové a hodnotové podle toho, jak přistupujeme k posouzení ekvivalence dvou instancí. Instance hodnotových datových typů reprezentují nějakou hodnotu a jejich dvě instance považujeme za ekvivalentní, pokud obě reprezentují stejnou hodnotu. Naproti tomu instance odkazových datových typů reprezentují nějaké jedinečné entity a jsou ekvivalentní pouze samy se sebou.

To, jestli je daný typ odkazový či hodnotový, poznáme snadno podle toho, definuje-li vlastní verzi metody `equals(Object)`. Odkazové datové typy se spokojí s verzí zděděnou od třídy objekt, která je definovaná tak, že (stejně jako operátor `==`) vracejí `true` pouze v případě, kdy porovnáváme instanci samu se sebou. V opačném případě vracejí `false`. U odkazových datových typů bývá tato informace opravdu tím, na co se chceme zeptat. U hodnotových datových typů nás ale většinou nezajímá porovnání instancí, ale porovnání jimi reprezentovaných hodnot, a proto musejí přebít zděděnou verzi verzí vlastní.

Tato poznámka je určena pro čtenáře, kteří mají zkušenosti s programováním v prostředí .NET. Tam totiž zavádějí terminologický guláš, protože typy nerozlišují podle toho, zda reprezentují hodnotu, ale podle způsobu alokace jejich instancí v paměti. Jako hodnotové datové typy označují typy, které se nealokují na haldě, ale na zásobníku návratových adres. To ale obecně nemusí být hodnotové typy, protože nemusejí představovat žádnou hodnotu. Tyto typy bychom mohli označit spíše jako kopírované, protože se u nich přiřazení realizuje zkopírováním daného objektu, a ne zkopírováním pouhého odkazu.

1.10 Návrhové vzory

V minulém dílu jsme se seznámili s několika návrhovými vzory. V tomto dílu k nim přidáme další. Připomenu, že návrhové vzory jsou programátorské ekvivalenty matematických vzorečků, v nichž místo čísel vystupují objekty a datové typy. Jsou to doporučení, jak vyřešit některé často se vyskytující úlohy.

Zopakujme si alespoň ve stručnosti ty, které byste měli z minulého dílu znát (uvádím je v abecedním pořadí). Neznáte-li je, pokuste se před dalším čtením doplnit své znalosti.

Jedináček (*singleton*)
řeší problém třídy, která má mít právě jednu instanci

Jednoduchá tovární metoda (*simple factory method*)
je statická metoda vracející jako svoji funkční hodnotu instanci třídy, v níž je definována. Odstraňuje některé nevýhody konstruktoru, a proto ji některé datové typy používají místo konstruktoru.

Knihovní třída (*library class, utility*)
je třída, která nemá žádné instanční členy. Definuje pouze statické konstanty a metody. Protože nepotřebuje instance, je definována tak, aby její instance nešlo vytvořit.

Posluchač (*listener*)
navrhuje efektivní řešení situace, kdy jeden či více objektů čeká na výskyt nějaké události (např. stisk tlačítka), aby na ni mohly zareagovat.

Pozorovatel (*observer*)
je pouze jiný název pro návrhový vzor *Posluchač*.

Prázdný objekt (*null object*)
představuje nesmyslnou instanci své mateřské třídy a používá se v situacích, kdy metoda nemůže vrátit žádnou platnou instanci. Mnohé metody vracejí v takových situacích prázdný odkaz (`null`), ale tomu se pak nedají posílat zprávy a tím se komplikuje okolní program. Alternativní způsob řešení tohoto problému si naznačíme v podkapitole *30.3 Datový typ* `Optional<T>` na straně 639.

Prototyp (*prototype*)
nabízí alternativní způsob vytváření nových instancí jako kopií instancí již existujících.

Prostředník (*mediator*)
řeší problém přílišného množství vazeb mezi jednotlivými třídami zavedením prostředníka, který komunikaci mezi třídami či jejich instancemi zprostředkuje.

Služebník (*servant*)
řeší problém, jak dodat dodatečnou společnou funkcionalitu skupině tříd, aniž bychom museli do každé přidávat téměř shodnou metodu.

Statická tovární metoda (*static factory method*)
je pouze jiný název pro návrhový vzor *Jednoduchá tovární metoda*.

Šablonová metoda (*template method*) doporučuje definovat v předku metodu i v případě, že je v ní potřeba použít činnosti, které budou definovat až potomci. Tyto činnosti se deklarují jako abstraktní metody s tím, že potomek je přebije vlastními, konkrétními metodami, které daná (šablonová) metoda předka použije.

Výčtový typ (*enumerated type, enumeration, enum, factor*)
je datový typ, jenž má předem dané instance, které není možno v průběhu programu přidat, ubrat ani změnit.

1.11 Modul × komponenta × knihovna × framework

Knihovny podprogramů se používají od začátku programování. S nástupem objektově orientovaného programování se zrodila i nová kategorie pomocných programů, pro které se vžilo označení framework.

Paralelně se začalo hovořit o modulech a modulárním programování, které následně dospělo do vyšší etapy komponentového programování.

Protože se s těmito termíny často setkáte (a to nejen v této učebnici) a protože v nich řada programátorů nemá zcela jasno, trochu je rozebereme.

Modul

Modul je logicky samostatná část programu určená k plnění dané funkce. Modul se snaží vyřešit vše sám a minimalizovat komunikaci s ostatními moduly s výjimkou případů, kdy některý z okolních modulů řeší nějakou jednodušší funkci, která je pro daný modul užitečná. Vzájemnou závislost modulů se ale snažíme minimalizovat.

Modul se může sám skládat z podmodulů, které plní dílčí funkce většího modulu a které se zase snažíme navrhnout vzájemně nezávislé.

Definice jazyka Java považuje za ekvivalenty klasických modulů balíčky. Programátoři, kteří přecházejí na objektové programování ze starších paradigmat (např. strukturované programování), často uvádějí jako příklady malých modulů i třídy (třída je podle nich modul obsahující sadu definic metod). Jenomže starší paradigmata neumožňovala s dostatečnou efektivitou vytvářet tak rozsáhlé programy, takže i granularita vnímání programátorů byla menší. Objektoví programátoři většinou považují za modul balíček či skupinu balíčků.

Komponenta

Komponenta je modul, který je možno použít samostatně, tj. nezávisle na aplikaci, pro niž původně vznikl. Komponenta má definované rozhraní a deklaruje požadavky na rozhraní komponent, s nimiž má spolupracovat. Se svým okolím komunikuje pouze prostřednictvím tohoto rozhraní. (Toto rozhraní nemusí(!) být definováno jako interfejs, i když v Javě tomu tak většinou bývá.)

Důležitou vlastností komponenty je, že je nahraditelná jinou komponentou se stejným rozhraním a funkcionalitou, a to nejenom v době návrhu, ale i za běhu programu. V grafickém uživatelském rozhraní můžete např. vyměnit seznam položek za přepínač. Každá z těchto komponent vypadá jinak, ale jejich funkčnost je stejná.

Další důležitou vlastností komponent je, že jsou do jisté míry schopny se přizpůsobit potřebám svého uživatele, aniž by bylo nutno dělat zásahy do jejich kódu. Jinými slovy: jsou konfigurovatelné. Zůstaňme u grafického uživatelského rozhraní, které mívá jako jednu ze svých komponent tlačítko. Tato komponenta je definovaná v samostatné knihovně. Nicméně je definovaná tak, že jí program může jednoduše zadat, jak má upravit svůj vzhled, jak má reagovat na stisk či přejetí myší apod. To vše bez zásahu do kódu komponenty.

Jako příklad velice jednoduchých komponent jsou někdy uváděny třídy, resp. jejich instance. Instanci jedné třídy můžete nahradit instancí jiné třídy se stejným rozhraním. Současně ji můžete prostřednictvím přístupových metod do jisté míry konfigurovat, aniž byste museli měnit její kód.

Knihovna

Knihovnou je sada pasivních entit (pasivních proto, že čekají, až je někdo použije) určených k použití okolním programem. V dřívějších dobách byly těmito entitami procedury a funkce a později se jimi stávaly i moduly a případně i komponenty.

Každý člen knihovny poskytuje nějakou službu, kterou lze využít. Organizace celého řešení je přitom zcela v rukou daného uživatele (uživatelem bývá většinou nějaký program).

Jako příklad knihovny by mohl sloužit např. balíček `util` projektu, s nímž jsme pracovali v minulém dílu. Ten obsahoval několik tříd, jejichž služeb jsme mohli využít, aniž bychom se jim museli nějak významně přizpůsobovat.

Framework

Framework se podobá knihovně s několika speciálními vlastnostmi:

- Organizace řešení nespočívá na bedrech uživatele, ale využitím principu inverze závislosti (viz první díl, strana 165, podkapitola *8.8 Nová koncepce* projektu) je vše v rukou frameworku, jemuž se musí jeho uživatel přizpůsobit.
- Framework má definované smysluplné implicitní chování.
- Framework není obecně modifikovatelný. Přesněji řečeno: veškeré požadavky na úpravu či rozšíření jeho funkčnosti by měly být realizovatelné bez zásahu do jeho kódu.
- Framework je ale rozšiřitelný. Jeho funkčnost lze rozšiřovat či upravovat přidáním vhodného kódu, který se požadovaným způsobem naváže na původní kód daného frameworku.

Příkladem jednoduchého frameworku by mohl být správce plátna (v novém projektu instance třídy `CanvasManager`) spolu s instancemi spolupracujících tříd. Chtějí-li objekty využívat služeb správce plátna a býti zobrazovány v aplikačním okně, musí se mu přizpůsobit tím, že implementují interfejs `IPaintable` a přihlásí se do správy správce. Ten pak sám rozhodne, kdy se má ten který přihlášený objekt nakreslit.

Framework má rozumné implicitní chování, i když po něm nic nechceme (`CanvasManager` zobrazí prázdné plátno). V minulém dílu jsme si navíc vyzkoušeli, že vhodnou definicí tříd implementujících požadované interfejsy lze dosáhnout zajímavých výsledků.

Protože z jistého zorného úhlu není mezi knihovnou a frameworkem žádný podstatný rozdíl, nebudu v dalším textu tyto termíny úpěnlivě odlišovat a budu-li hovořit o knihovnách, bude se to vztahovat i na frameworky. Kdyby tomu tak nemělo být, výrazně na to upozorním.

1.12 Změny šablon

V prvním dílu jsem zavedl šablony, které definovaly výchozí podobu vytvářených zdrojových souborů. Při práci se studenty a řešení jejich problémů ale časem vykrystalizoval jeden z důvodů, proč mají začátečníci často potíže s konstrukcemi týkajícími se atributů a metod třídy. Tímto důvodem je zaužívaný způsob zápisu zdrojových programů, při němž se do zdrojového kódu zapisují odděleně atributy a metody a v rámci jejich sekcí se pak definují zvlášť členy třídy a zvlášť členy instancí. To vede k promíchání částí zdrojového kódu definujících vlastnosti a schopnosti objektu třídy a vlastnosti a schopnosti jejích instancí.

Se studenty jsme se shodli, že by pro ně bylo výhodnější (a srozumitelnější), kdybychom ve zdrojovém kódu napsali zvlášť vše, co se týká objektu třídy, a pak odděleně vše, co se týká instancí této třídy. Podle tohoto pravidla jsem proto upravil šablony pro všechny následující kurzy, a tím pádem i pro tuto knihu.

Zdrojový kód prázdné třídy definované podle nové šablony si můžete prohlédnout ve výpisu 1.1.

Jak se můžete přesvědčit, části věnované třídním atributům a metodám jsou v ní výrazně odděleny od částí věnovaných instančním atributům a metodám. Obě pak mají téměř shodnou strukturu. O interních datových typech, jimž je věnována poslední sekce nadepsaná `NESTED DATA TYPES`, jsme ještě nehovořili. Budeme se jim podrobně věnovat v kapitole *19 Interní datové typy* na straně 397.

Výpis 1.1: *Prázdná třída vytvořená podle standardní šablony*

```
/*******************************************************************************
 * Instance třídy {@code ClassTemplate} představují
 *
 * @author  Rudolf PECINOVSKÝ
 * @version 0.00.0000 – 20yy-mm-dd
 */
public class ClassTemplate
{
//== CONSTANT CLASS ATTRIBUTES ================================================
//== VARIABLE CLASS ATTRIBUTES ================================================



//##############################################################################
//== STATIC INITIALIZER (CLASS CONSTRUCTOR) ====================================
//== CLASS GETTERS AND SETTERS ================================================
//== OTHER NON-PRIVATE CLASS METHODS ==========================================
//== PRIVATE AND AUXILIARY CLASS METHODS ======================================


//##############################################################################
//== CONSTANT INSTANCE ATTRIBUTES =============================================
//== VARIABLE INSTANCE ATTRIBUTES =============================================


//##############################################################################
//== CONSTUCTORS AND FACTORY METHODS ==========================================

    /***************************************************************************
     *
     */
    public ClassTemplate()
    {
    }


//== ABSTRACT METHODS =========================================================
//== INSTANCE GETTERS AND SETTERS =============================================
//== OTHER NON-PRIVATE INSTANCE METHODS =======================================
//== PRIVATE AND AUXILIARY INSTANCE METHODS ====================================


//##############################################################################
//== NESTED DATA TYPES ========================================================
}
```

Šablona interfejsů má také novou podobu. Tu vám ale prozatím neukážu, protože Java 8 výrazně rozšiřuje jejich dosavadní účel. Se šablonou interfejsů vás proto seznámím až v podkapitole *14.6 Šablona interfejsů* na straně 306, kdy už budete potřebné novinky znát, a budete proto chápat význam jednotlivých jejích sekcí.

Stejně tak si budete muset počkat na šablonu testovací třídy, protože v ní začneme používat jednu zajímavou pomocnou třídu. Šablonu testovací třídy vám proto představím až v podkapitole *7.3 Šablona testovací třídy* na straně 166.

1.13 Knihovna CanvasManager

V tomto dílu budeme dále pracovat s knihovnou `CanvasManager`, s níž jsme pracovali v minulém dílu. Budu přitom předpokládat, že knihovnu již znáte, takže se na její třídy budu poměrně často odvolávat. Ti, kteří nečetli minulý díl, anebo již vlastnosti jejích jednotlivých tříd zapomněli, najdou potřebné informace v její programátorské dokumentaci, která je součástí stejnojmenného projektu.

1.14 Shrnutí – co jsme se naučili

- Přehled předpokládaných znalostí, které budete při studiu této knihy potřebovat, najdete na stránce knihy v samostatném souboru.
- Deklarace je část kódu, v níž předem oznamujeme některé vlastnosti vytvářeného programu.
- Definice je část kódu, v níž něco doopravdy vytváříme.
- Každý program je simulací reálného nebo virtuálního světa.
- Jako objekt označujeme v programu kolekci dat reprezentující odpovídající skutečnost z onoho simulovaného světa.
- Objekty v programu mohou reprezentovat cokoliv, co můžeme pojmenovat, tj. cokoliv, co můžeme označit podstatným jménem.
- Termínem *zpráva* označujeme požadavek, který zasílá jeden objekt druhému. Odesílatel zprávy přitom očekává od adresáta reakci odpovídající kontraktu dané zprávy.
- Termínem *metoda* označujeme kód definující, jak bude daný objekt reagovat na zaslanou zprávu.

- Některým zprávám je přiřazeno více metod, mezi nimiž si oslovený objekt může vybrat.
- Schopnost objektů reagovat na stejnou zprávu různě označujeme jako **polymorfismus**.
- V knize budu důsledně odlišovat termíny *rozhraní* jako obecnou vlastnost libovolné entity v programu a *interfejs* jako označení programové konstrukce specifikující požadované rozhraní, a případně i doporučující implicitní implementaci.
- Interfejsy představují datové typy reprezentující rozhraní instancí tříd, které tyto interfejsy implementují.
- Modul je logicky samostatná část programu určená k plnění dané funkce.
- Modul se může sám skládat z podmodulů.
- Komponenta je modul, který je možno použít samostatně.
- Komponenta by měla být nahraditelná jinou komponentou se stejným rozhraním a funkcionalitou.
- Komponenty jsou konfigurovatelné, jejich funkcionalitu je možné do jisté míry upravovat bez zásahu do jejich kódu.
- Knihovna je sada pasivních entit určených k použití okolním programem.
- Framework je knihovna se speciálními vlastnostmi:
  - Použitím principu inverze závislosti přebírá framework řízení chování programu.
  - Má definované smysluplné implicitní chování.
  - Není modifikovatelný, ale je rozšiřitelný.
- Oproti předchozímu dílu se změnilo pořadí sekcí standardní šablony; nyní je v ní pohromadě veškerý kód týkající se třídy, který je konstruktory a továrními metodami oddělen od kódu týkajícího se instancí.

Kapitola 2
Vývojové prostředí NetBeans

Co se v kapitole naučíte

Seznámíte se s profesionálním vývojovým prostředím *NetBeans*. Nejprve vám ukážu, jak se instaluje. Potom v něm otevřeme existující projekt a předvedeme si, jak se pracuje s panely. Ve zbytku kapitoly vám budu vysvětlovat, jak upravit konfiguraci prostředí podle potřeby. Při té příležitosti se dozvíte řadu informací o jeho vlastnostech.

V této kapitole si po instalaci prostředí otevřeme projekt **N202a_Library** (ten si stáhnete prostřednictvím generátoru, o němž jsem mluvil v úvodu), na němž si předvedeme některé základní vlastnosti vývojového prostředí.

V minulém dílu jsme pracovali ve vývojovém prostředí *BlueJ*. Řekli jsme si, že patří mezi nejjednodušší IDE, což je zkratka z anglického *Integrated Development Environment* – integrované vývojové prostředí, přičemž adjektivum *integrovaný* označuje, že daný program integruje více nástrojů dohromady – editor, překladač, ladící program a řadu dalších.

Prostředí *BlueJ* je optimalizované na výuku a na to, aby se s jeho pomocí studenti naučili přemýšlet objektově. Pro tvorbu složitějších programů to však není optimální volba – pro tu je vhodné použít některé z profesionálních IDE.

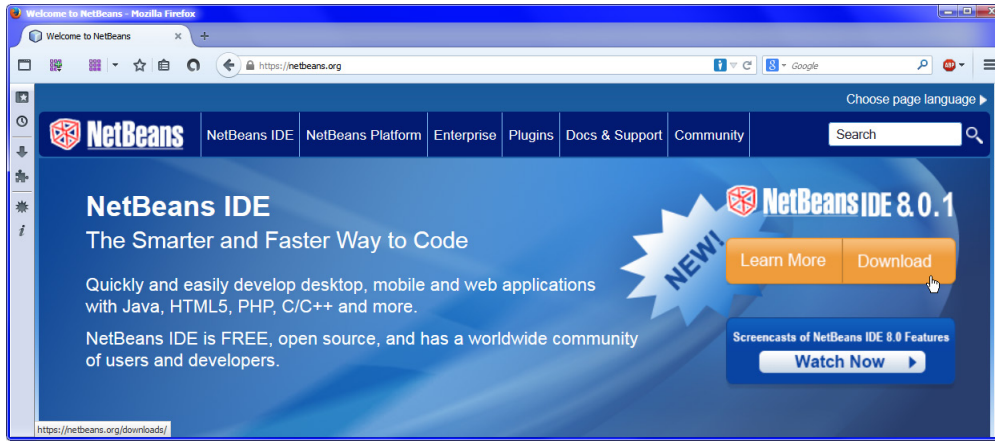
Pro vývoj programů v jazyce Java se nejvíce používají tři prostředí: *Eclipse*, *IntelliJ IDEA* a *NetBeans*. Tato prostředí jsou si vzájemně velmi podobná, a jakmile přijde některé z nich se zajímavou novinkou, zanedlouho ji do svého repertoáru začlení i ta ostatní. Zkušenost ale ukázala, že pro výuku je nejvhodnější poslední z nich, protože předchozí dvě nenabízejí některé funkce, které při tvorbě a následné demonstraci výukových programů využívám.

Jak jsem již ale řekl v úvodu, zvládnutí profesionálního vývojového prostředí je zhruba stejně náročné jako zvládnutí programovacího jazyka. Proto bude zbytek první části věnován seznámení se základními vlastnostmi *NetBeans*. Můžete samozřejmě dát přednost jinému IDE, ale základní popisované principy budou u všech velmi podobné. Bude se jen maličko lišit jejich ovládání a některé další detaily.

2.1 Instalace

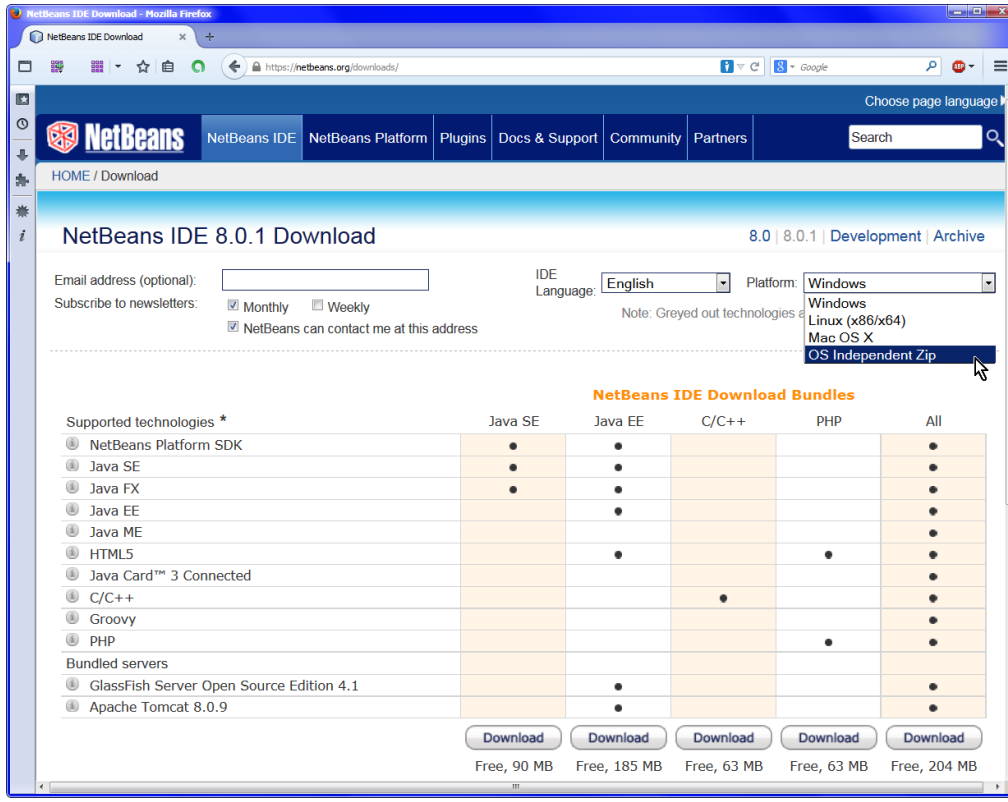
Vývojové prostředí *NetBeans* je profesionální vývojové prostředí, které je k dispozici zdarma. Můžete si je stáhnout z jeho stránek.

1. Otevřete stránku http://netbeans.org/ a stiskněte tlačítko **Download** umístěné vpravo pod názvem produktu (viz obrázek 2.1).



Obrázek 2.1
Domovská stránka produktu NetBeans

2. Otevře se stahovací stránka, na které si budete moci vybrat jazyk lokalizace staženého prostředí (k dispozici je angličtina, japonština, čínština, portugalština a ruština) a operační systém (na obrázku 2.2 je tato nabídka rozbalená). V seznamu dostupných operačních systémů bych rád upozornil na poslední položku (na obrázku je zvýrazněna), která umožní stáhnout ZIP, jenž můžete rozbalit a spustit kdekoliv. Musíte se však smířit s tím, že tento instalační soubor je, oproti instalaci pro konkrétní operační systém o 70 až 87 MB větší (záleží na tom, který z nabízených systémů berete jako referenční) a některé drobnosti (např. instalaci knihovny *JUnit*) musíte realizovat „ručně“.



Obrázek 2.2

Stránka, na níž si vybíráte možné konfigurace instalovaného programu

3. Když už jste si vybrali jazyk a operační systém, musíte ještě zvolit, jakou instalační sadu chcete stáhnout, přičemž u každé z nabízených sad hned vidíte, co vše je příslušná sada schopna podporovat (viz obrázek 2.2) a jak velký je její instalační soubor. (Jak jistě sami odhadnete, pro náš kurz bude stačit ta nejjednodušší, cca 90MB verze.) Stiskněte proto tlačítko pod vybraným sloupcem.
4. Zadaná instalační sada se stáhne do vašeho počítače. Kam, to záleží na nastavení vašeho prohlížeče.

Instalace pro Windows

V dalším výkladu budu pokračovat popisem instalace pro *Windows*, protože tento operační systém používá přes 80 % studentů a účastníků mých kurzů. Ti z vás, kteří si stáhli onen univerzální ZIP, mohou výklad přeskočit, ti, kteří budou instalovat pod jiným operačním systémem, mi musejí odpustit případné odchylky. Takže pokračujeme:

5. Spusťte stažený instalační soubor.

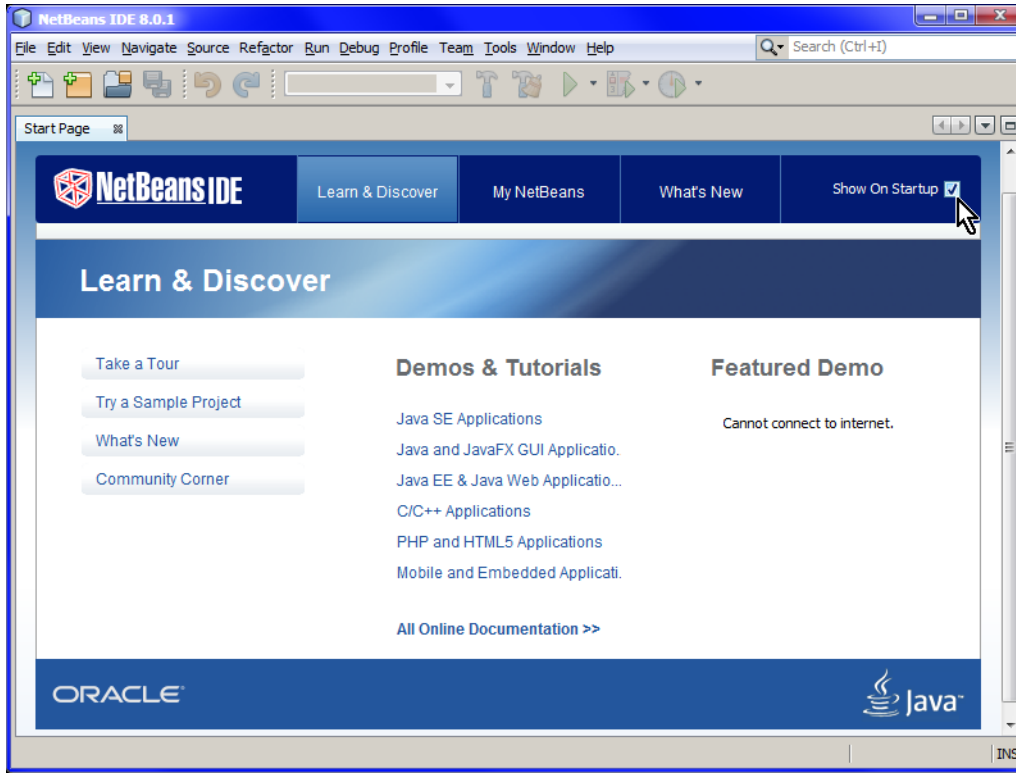
6. Program zanalyzuje váš počítač, a bude-li vše v pořádku, otevře okno instalátoru, v němž vám oznámí, co se chystáte instalovat a kolik diskového prostoru k tomu budete potřebovat. Stiskněte **Next>**.
7. Otevře se okno s licenčním ujednáním. Přečtěte si je, zaškrtněte políčko označující souhlas (předpokládám, že budete ochotni toto ujednání akceptovat) a stiskněte **Next>**.
8. Objeví se další okno s licenčním ujednáním – tentokrát na knihovnu *JUnit*, s níž jsme se již seznámili v minulém dílu. Tentokrát si budete moci vybrat, zda chcete knihovnu instalovat. V knize předpokládám, že nastavíte horní položku přepínače, čímž oznamujete, že licenční ujednání akceptujete a knihovna se smí instalovat. Nastavte ji a stiskněte **Next>**.
9. Otevře se okno se dvěma vstupními poli: v horním zadáváte, kam chcete *NetBeans* instalovat, v dolním oznamujete, kde je instalovaná verze Javy (přesněji JDK), pod kterou budete *NetBeans* spouštět. Obě políčka jsou předvyplněná.

   Nechce-li se vám přemýšlet nad umístěním programu, můžete rovnou jít dál. Jak už jsem ale říkal v minulém dílu, já jsem se rozhodl všechny programy v Javě umísťovat do složky **Java** v kořenové složce mého systémového disku. Oddělí se tak programy spouštěné přímo nad platformou mého operačního systému od programů spouštěných nad platformou Java.

   Rozhodnutí nechám na vás. Poté, co zadáte obě umístění, stiskněte tlačítko **Next>**.
10. Otevře se závěrečné okno průvodce instalací, v němž vám instalátor oznamuje, kam bude *NetBeans* instalovat, že se knihovna *JUnit* nainstaluje po prvním spuštění, že se budou automaticky zjišťovat případné aktualizace a kolik celá instalace zabere místa na disku. V tuto chvíli máte poslední šanci se stiskem tlačítka **<Back** vrátit do některého z předchozích oken a opravit svoje zadání. Jste-li se zadáním spokojeni, stiskněte tlačítko **Install**.
11. Otevře se instalační okno, v němž vás bude program průběžně informovat o aktuálně instalovaných souborech a o tom, jak je s instalací daleko. Pro případ, že byste si to s instalací rozmysleli, je připraveno tlačítko **Cancel**, ale předpokládám, že necháte instalaci doběhnout do konce.
12. Po skončení instalace se otevře ukončovací okno, v němž vám instalátor oznamuje, že instalaci úspěšně dokončil, a současně vám prozradí, jak lze program spustit. Zároveň zde najdete zaškrtávací políčko, jehož zaškrtnutím souhlasíte s tím, aby program posílal vývojovému týmu stručné anonymní informace o používaných funkcích, které budou podkladem pro rozhodnutí, o co by mohla být vylepšena příští verze. Já je zaškrtávám, ale vy se můžete rozhodnout jinak.

2.2 První spuštění

Po spuštění programu chvíli uvidíte startovací okno, které u svého spodního okraje zobrazuje, jak je natahování programu daleko. Po několika vteřinách (nebo desítkách vteřin – záleží na rychlosti počítače a použitých disků) se otevře aplikační okno se startovní stránkou *Start Page*, která má tři karty (viz obrázek 2.3).



Obrázek 2.3
Uvítací stránka aplikačního okna

- Karta **Learn & Discover** obsahuje hypertextové odkazy na stránky s nejrůznějšími tutoriály, návody, ukázkovými příklady a dalšími informacemi, z nichž byste se mohli poučit.
- Prostřední karta **My NetBeans** teď bude téměř prázdná, ale až začnete *NetBeans* používat, bude zobrazovat seznam naposledy otevřených projektů a instalovaných pluginů.
- Poslední karta **What's New** nabízí seznam posledních článků o *NetBeans*. Najdete zde odkazy na zprávy, tutoriály i různé blogy. Odkazy na této stránce vedou k informacím různých úrovní – některé jsou vhodné i pro začátečníky, jiné předpokládají jisté znalosti a zkušenosti.

Hlavní nevýhodou všech těchto výukových a informačních textů je (alespoň pro některé) to, že jsou všechny anglicky. Jak už jsem ale říkal v úvodu, slušně programovat se bez znalosti angličtiny nedá.

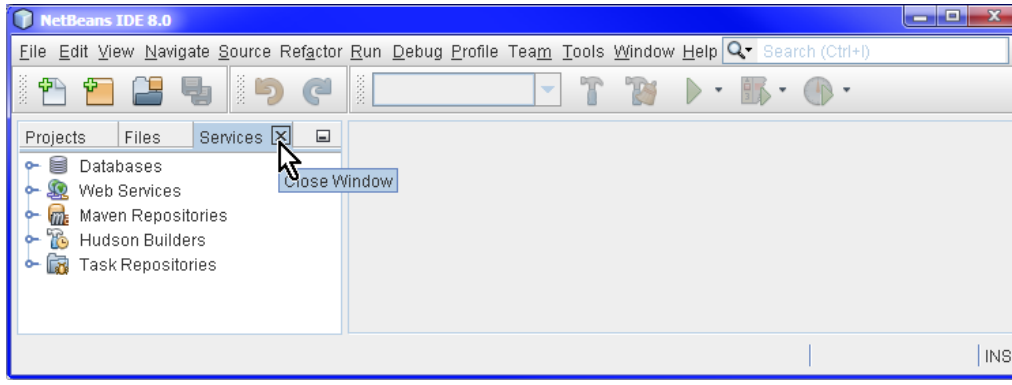
Na stránce je v pravém horním rohu připravené zaškrtávací políčko **Show On Startup** (na obrázku 2.3 na něj ukazuje kurzor myši), jehož „odzaškrtnutím" oznámíte, že se při příštím startu stránka nemá zobrazovat. Kdybyste si to později rozmysleli a chtěli tuto stránku opět zobrazit, stačí v nabídce **Help** zadat příkaz **Start Page** a stránka se okamžitě zobrazí.

My ale teď na nějakou dobu oželíme její dobrodiní. Zrušte proto zaškrtnutí políčka **Show On Startup** a startovní stránku zavřete.

2.3 Aplikační okno, panely a karty

Po zavření startovní stránky se objeví prázdné aplikační okno (zmenšené je na obrázku 2.4) rozdělené na dva panely. Pravý z nich je prázdný, kdežto levý uchovává tři karty[3] se záložkami v záhlaví.

Pravá záložka reprezentuje kartu **Services**. Tu využijí až pokročilí vývojáři, takže ji nebudeme v průběhu celého kurzu potřebovat, a můžeme ji proto zavřít. Před tím si ale vyzkoušíme, co vše můžeme s takovou kartou dělat.

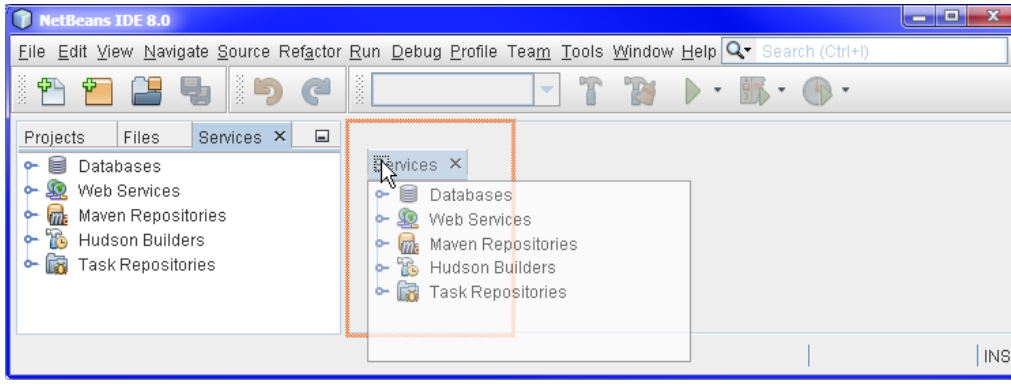


Obrázek 2.4
*Zavření karty **Services** v levém panelu aplikačního okna*

Najeďte myší na její záložku a uchopte ji. (Pozor ale, abyste ji místo toho klepnutím na zavírací ikonu nezavřeli, jak se k tomu chystá myš na obrázku 2.4.) Když nyní popojedete s myší, bude ji karta poslušně následovat. Jak s ní budete jezdit

[3] Nabídky, dokumentace i nápověda hovoří sice o oknech, ale pak musíte říkat, že v okně je okno se záložkou a jiná podivná tvrzení. S kartami bude výklad srozumitelnější.

po aplikačním okně, tak se budou na různých místech objevovat červené rámečky, které budou naznačovat, kam by se karta usadila, kdybyste ji v daném okamžiku pustili (viz obrázek 2.5).



Obrázek 2.5
*Přesun okna **Services** do alternativního umístění*

Při popojíždění může nastat několik situací:

- Přejedete-li s myší k okraji nějakého panelu (je jedno, jestli to bude panel, v němž se okno nachází, nebo nějaký jiný), zobrazí se obdélníkový rámeček obdobný tomu, který vidíte na obrázku 2.5, kde jsem s ním přejel až do sousedního panelu. Pak se panel, v němž se právě nacházíte, rozdělí na dva a část naznačenou rámečkem zabere nový panel, do něhož bude umístěna přesouvaná karta.
- Přejedete-li s myší do oblasti záložek, zobrazí se „zubatý“ rámeček naznačující záložku. Pak karta zůstane ve stejném panelu, jenom její záložka bude mezi záložkami daného panelu na jiném místě.
- Odjedete-li myší mimo aplikační okno, otevře se pro ni nové okno. Do tohoto okna se umístí naše karta se záložkou. K ní pak můžete v budoucnu přidávat i další karty. Okno už ale nepůjde rozdělit na několik panelů.

Této možnosti můžete využít, používáte-li počítač s několika monitory. Na každém monitoru pak můžete mít jedno či více oken a v nich libovolné množství karet. Já to využívám zejména při ladění. Až si o něm budeme povídat, tak vám to připomenu.

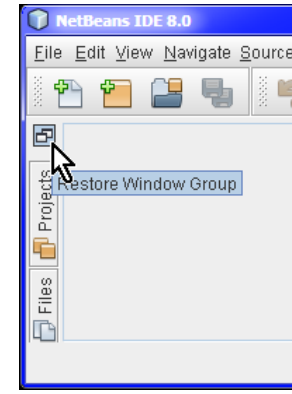
Změny rozměrů panelů

Vyzkoušejte si různé možnosti umisťování karty. Při té příležitosti si vyzkoušejte, že vzájemnou velikost panelů můžete změnit tak, že myší najedete na dělící hranu, uchopíte ji a přesunete. To platí jak pro panely umístěné vedle sebe, tak pro panely umístěné nad sebou.

Minimalizace a obnovení panelů a karet

Občas se stane, že nám zrovna nějaký z panelů u okraje plátna překáží a potřebovali bychom mít k dispozici místo, které zabírá. Jednou z možností, jak takovouto situaci řešit, je změnit rozměr daného panelu, jak jsme si říkali před chvílí. Druhou možností je panel dočasně (nebo i trvale) minimalizovat.

Na obrázku 2.5 jste si jistě všimli, že vpravo vedle karty **Services** je připravené minimalizační tlačítko. Pokud je stisknete, panel zaparkuje do okraje aplikačního okna, u kterého právě sídlí. Záložky karet se seřadí podél daného okraje. Klepnutím na kteroukoliv z nich příslušnou kartu rozbalíte a dalším klepnutím ji opět sbalíte.



Obrázek 2.6
Znovuotevření zaparkovaného panelu

Vedle záložek se současně objeví obnovovací tlačítko (viz obrázek 2.6). Když na ně klepnete, panel zaujme svoji původní pozici i rozměr a záložky karet z tohoto panelu se opět vrátí do jeho záhlaví.

Další možnosti

Panely nabízejí ještě další možnosti uspořádání, o nichž jsem nehovořil. Nechtěl jsem vás totiž zbytečně zdržovat, protože vím, že už se nemůžete dočkat, až začneme opět programovat. Probral jsem proto zrychleně pouze ty možnosti, které budeme za chvíli potřebovat.

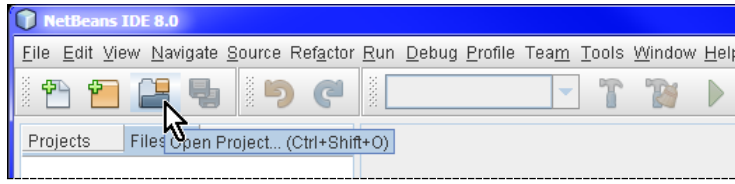
Koho další možnosti zajímají, tak na ně při troše experimentování jistě přijde. Vše si vyzkoušejte a až si dostatečně pohrajete, tak kartu **Services** zavřete.

2.4 Otevření existujícího projektu

Obdobně jako v *BlueJ*, i v *NetBeans* jsou vyvíjené programy uspořádány do projektů. Jeden program může být rozdělen do několika projektů, aby na něm mohlo snáze pracovat několik vývojářů. Nicméně součástí *NetBeans* jsou i nástroje, které umožnují dostatečně efektivní práci několika vývojářů na jediném projektu.

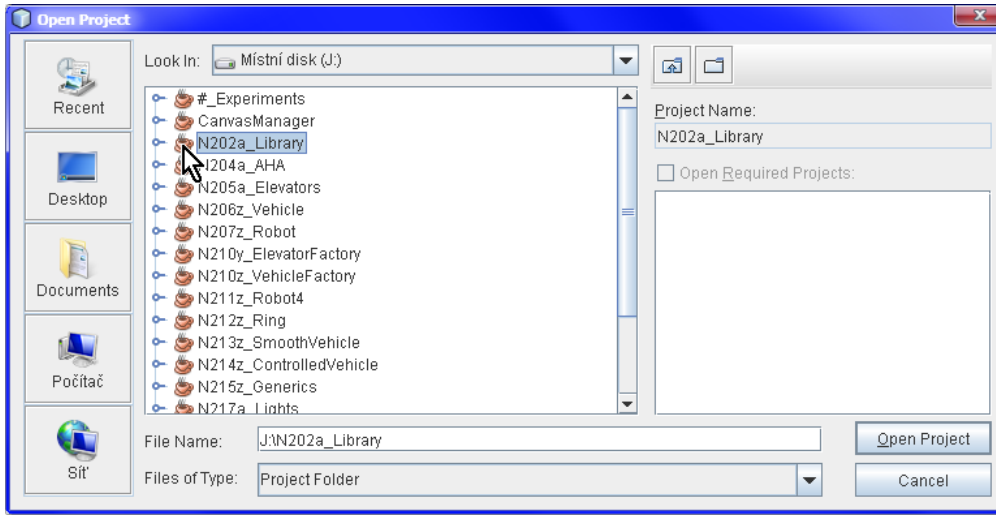
Otevřeme svůj první projekt v *NetBeans*. Abychom se nejprve dozvěděli něco o *NetBeans*, nebudeme otevírat nový, prázdný projekt, v němž bychom museli teprve definovat jeho programy (i na to časem dojde), ale otevřeme předpřipravený projekt s funkčním programem. Postupně se seznámíme s jeho součástmi a ukážeme si, jaké výhodné funkce pro práci nám *NetBeans* nabízejí.

Pojďme tedy na to. Předpokládám, že jste již spustili generátor a na zvolené místo uložili jeho projekty (pro tuto kapitolu bude stačit, budete-li mít uložený projekt **N202a_Library).** Existující projekt otevřete buď klepnutím na ikonu **Open Project** (standardně je umístěna jako třetí zleva – viz obrázek 2.7), nebo zadáním příkazu **File → Open Project**, anebo stiskem klávesové zkratky CTRL+SHIFT+O.



Obrázek 2.7
Žádost o otevření existujícího projektu

NetBeans otevřou dialogové okno pro otevření existujícího projektu (viz obrázek 2.8). V něm buďto zadáte do vstupního pole **File name** cestu ke složce s projektem, anebo tuto složku najdete prostřednictvím polí panelu se stromem složek. Ve stromu složek poznáte složku s projektem podle ikony představující šálek kávy. Vyberte složku s projektem **N202a_Library** a své zadání potvrďte stiskem tlačítka **Open Project**.



Obrázek 2.8
Dialogové okno pro otevření existujícího projektu

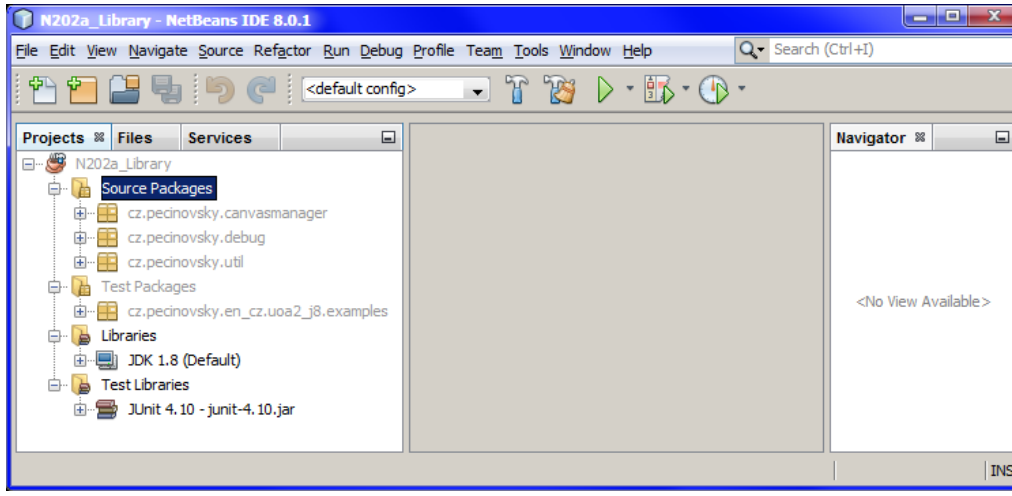
V *NetBeans* se na kartě projektů objeví položka s právě otevřeným projektem. Pod panelem s kartou projektů se narodí další panel. V něm bude jediná karta, jejíž záložka nám prozradí, že se jedná o kartu navigátoru (za chvíli si prozradíme, co to znamená).

Toto uspořádání (tj. panel s kartou navigátoru pod panelem s kartou projektů) přetrvává z dob, kdy byly ještě používány monitory s poměrem stran 4:3. Protože předpokládám, že již používáte širokoúhlý monitor, doporučím vám uspořádání, které považuji za výhodnější: uchopte kartu navigátoru, a způsobem, který jste si před chvílí zkoušeli s kartou **Services**, ji přesuňte (a s ní i její panel) zcela vpravo, aby mezi jejím panelem a panelem s projekty zůstalo ještě volné místo (viz obrázek 2.9) – tam budeme za chvíli otevírat okna editoru.

Než se pustíme do analýzy struktury projektu, chtěl bych vás upozornit na jeden rozdíl oproti tomu, na co jste byli zvyklí v předchozím dílu. Prostředí *BlueJ*, s nímž jsme pracovali v předchozím dílu, umísťovalo všechny soubory do jedné složky. Profesionální vývojová prostředí umožňují umístění souborů do více složek a sama pak implicitně nastavují samostatný strom složek pro zdrojové soubory vlastní aplikace a další pro zdrojové soubory testů. K tomuto rozdělení se ještě několikrát vrátíme.

Vraťme se ke kartě projektů. Položka s projektem je prozatím sbalená. Když ji rozbalíte, najdete v ní několik složek:

- Složka Source Packages obsahuje zdrojové soubory daného projektu. Rozbalíte-li ji, najdete v ní tři podsložky odpovídající třem balíčkům (viz obrázek 2.9).

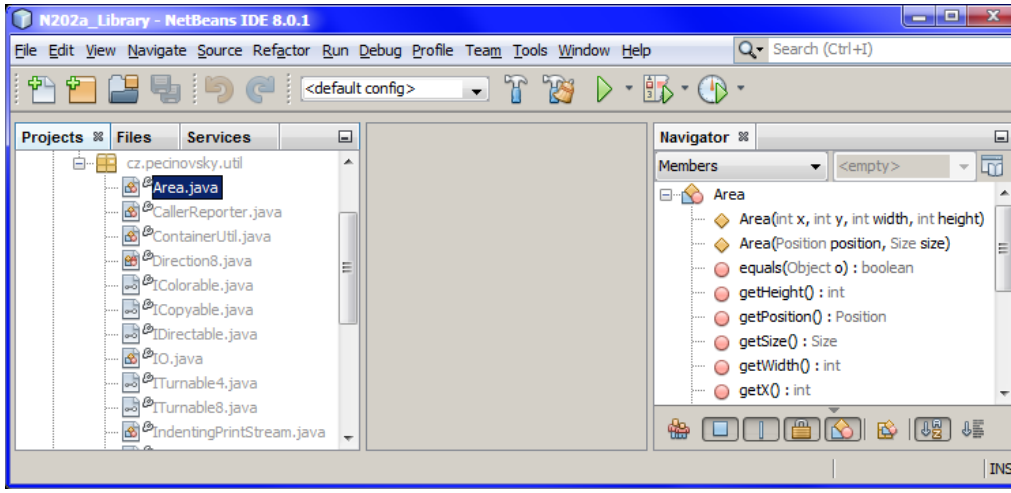


Obrázek 2.9
Rozbalování projektu v panelu projektů

- ☞ Složka Test Packages obsahuje testovací soubory projektu. Rozbalíte-li ji, najdete v ní jedinou podsložku odpovídající balíčku s testovacími třídami.
- ☞ Složka Libraries obsahuje seznam odkazů na potřebné knihovny, frameworky a jiné projekty, které daný projekt potřebuje ke své práci. Rozbalíte-li ji, najdete v ní jedinou podsložku odpovídající standardní knihovně. Tu můžete dále rozbalovat a analyzovat její jednotlivé součásti, ale já budu prozatím tuto možnost ignorovat a její analýzu ponechám vaší soukromé iniciativě.
- ☞ Složka Test Libraries obsahuje doplňkový seznam knihoven, frameworků a pomocných projektů potřebných pro funkci testů. Rozbalíte-li ji, najdete v ní jedinou podsložku odpovídající testovací knihovně (frameworku) *JUnit*, kterou jsme využívali již v minulém dílu, i když to prostředí *BlueJ* veřejně nijak neinzerovalo.

Pojďme se věnovat zdrojovým souborům projektu. Rozbalíte-li ve složce Source Packages druhou podsložku odpovídající balíčku `cz.pecinovsky.util`, najdete v ní jednotlivé zdrojové soubory daného balíčku. Klepnutím na položku požádáte panel navigátoru o zobrazení struktury daného souboru. Navigátor zobrazí definované třídy, atributy a metody tak, jak to ukazuje obrázek 2.10.

Poklepáním na položku zdrojového souboru nebo zadáním příkazu **Open** v její místní nabídce otevřete v panelu editoru kartu se zdrojovým kódem tohoto souboru.



Obrázek 2.10
Zobrazení obsahu souboru v panelu navigátoru

2.5 Navigátor a jeho ikony

Vraťme se ale k seznamu definovaných typů, metod a atributů zobrazovanému v panelu navigátoru. Protože u některých tříd je těchto položek opravdu požehnaně, umožňují ikony pod seznamem ovlivnit, co vše se bude zobrazovat. Význam jednotlivých tlačítek je následující:


Show inherited members – Zobrazování zděděných členů


Show fields – Zobrazování atributů


Show static members – Zobrazování statických členů


Show non-public members – Zobrazování neveřejných členů (není-li stisknuto, zobrazují se pouze členy s modifikátorem přístupu `public`).


Show inner classes – Zobrazování interních typových členů (budeme je probírat v kapitole *19 Interní datové typy* na straně 397).


Show Fully Qualified Names – Zobrazování názvů tříd s úplnou kvalifikací


Sort by Name – Členy jsou seřazeny dle abecedy


Sort by Source – Členy jsou seřazeny stejně jako ve zdrojovém kódu

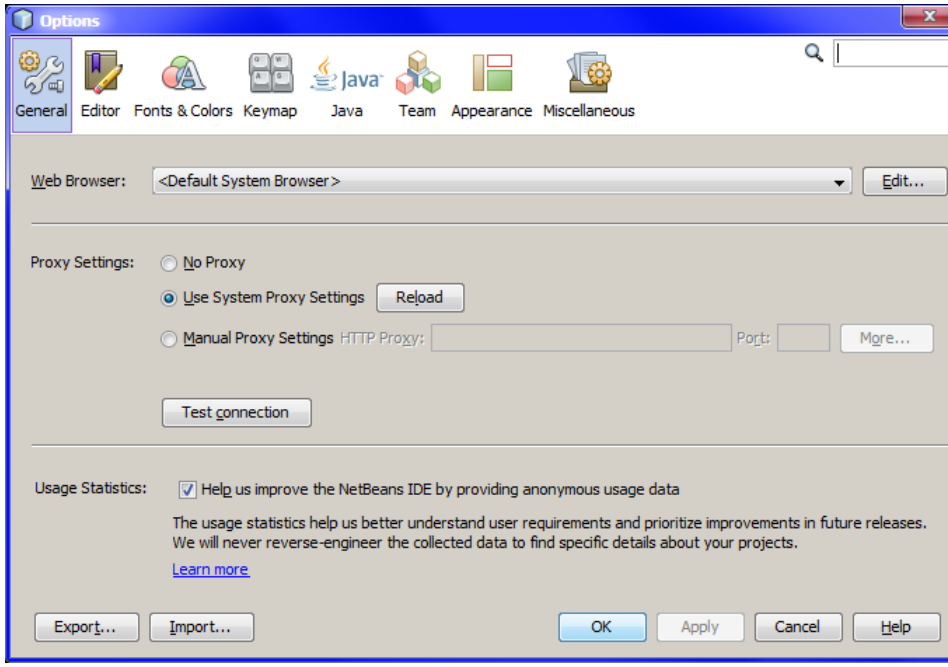
Poklepáním na položku v navigátoru se v panelu editoru zobrazí zdrojový kód s touto položkou. Pokud se ještě zdrojový kód nezobrazoval, otevře se jeho karta.

2.6 Úprava nastavení prostředí

Vítězně jsme se tedy dvěma způsoby probojovali ke zdrojovým souborům. Než si ale začneme vyprávět o práci s nimi, naučím vás, jak nastavovat některé vlastnosti vývojového prostředí *NetBeans*. Při té příležitosti se dozvíte o jeho vlastnostech a funkcích i řadu informací, které se vám budou při práci hodit.

Nebudu vám ale vysvětlovat všechny možnosti, které si můžete v *NetBeans* nastavit – to bychom se pak už v knize nedostali k ničemu jinému. Seznámím vás s několika nastaveními, která mohou být užitečná i začátečníkům. Zadejte proto příkaz **Tools** → **Options**, a otevřete tak dialogové okno (viz obrázek 2.11), s jehož pomocí můžete vlastnosti vývojového prostředí přizpůsobit svým požadavkům a potřebám.

Okno pro nastavování vlastností prostředí nabízí nastavení pro osm oblastí reprezentovaných ikonami v horním řádku. Pojďme si je postupně probrat.



Obrázek 2.11
Dialogové okno pro nastavování vlastností prostředí

2.7 General – obecná nastavení

Pro obecná nastavení (ikona **General** aktivovaná na obrázku 2.11) toho tu moc není. Nastavení internetového prohlížeče v seznamu **Web Browser** se uplatní při prohlížení vygenerované dokumentace API. Nechcete-li používat jiný než implicitně nastavený prohlížeč vašeho systému, můžete je ignorovat.

Zaškrtávací políčko **Usage Statistics** umožňuje dodatečně změnit vaše rozhodnutí o zasílání anonymních statistik vaší činnosti, o kterém jsme již hovořili v pasáži o instalaci *NetBeans*.

Zajímavá by pro vás mohla být tlačítka **Export** a **Import**. Ta umožňují uložit vaše aktuální nastavení do ZIP souboru (**Export**), resp. načíst obdržené nastavení (**Import**), a přizpůsobit tak svoje prostředí podle vzoru toho, kdo pro vás své nastavení exportoval.

Připravil jsem pro vás soubor s některými svými nastaveními, která by se vám mohla hodit. Projdeme si nejprve, co vše by mohlo být užitečné nastavit. Na konci vám pak v podkapitole *2.16 Export a import nastavení* na straně 72 ukážu, jak můžete importovat nastavení, která chcete převzít.

2.8 Editor – nastavení editoru

V oblasti **Editor** je k dispozici deset karet. Většinou je již nebudu zobrazovat – to si můžete udělat na počítači sami. Pouze vás seznámím s nejdůležitějšími možnostmi, které si na nich můžete nastavit.

Karta General

Na kartě obecných nastavení editoru najdete tři sekce:

Braces Matching

Zaškrtnutím políčka **Show outline** požadujete, aby se po vybrání závorky zvýraznila i její párová závorka a snáze jste tak odhalili, jestli máte závorky správně spárované, a pokud ne, tak která vám chybí či přebývá.

Zaškrtnutím políčka **Show tooltips** si objednáváte chování editoru pro případ, kdy je otevírací závorka bloku, v němž se právě nacházíte, mimo zobrazovanou oblast zdrojového textu. Při zaškrtnutí tohoto pole se v takovém případě u horního okraje okna editoru zobrazí začátek aktuálního bloku jako nápověda (tool tip).

Camel Case Behavior

Zaškrtnutím políčka v této sekci aktivujete příjemnou vlastnost, kterou bychom mohli v překladu označit jako *velbloudí vyhledávání*. Kdykoliv potřebujete zadat v programu nějaký identifikátor, stačí zapsat první písmeno a pokračovat jenom velkými písmeny daného identifikátoru. Editor vám pak nabídne seznam identifikátorů, které lze na daném místě použít a jejichž velbloudí notace odpovídá zadané.

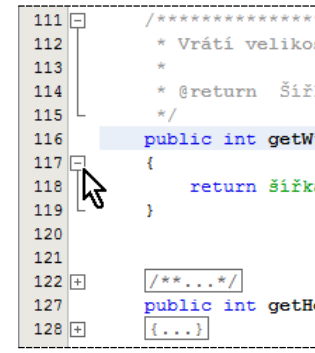
Budete-li chtít zadat identifikátor `metodaSeZvláštnímNázvem`, stačí zadat `mSZN` a seznam nabídnutých identifikátorů již bude pravděpodobně obsahovat pouze tuto metodu. Znaky samozřejmě nemusíte zadat všechny – tak, jak je budete postupně doplňovat, bude se seznam zkracovat.

Search

Poslední sekce umožňuje nastavit, jak bude editor při vyhledávání reagovat na stisk klávesy ENTER. Můžete buď přejít k dalšímu výskytu (**Enter to find next**), nebo ukončit vyhledávání (**Enter to finish search**).

Karta Folding

Termínem **Code Folding** se označuje skrývání (nebo chcete-li sbalení) částí kódu. Na této kartě nastavujete, zda mají *NetBeans* tuto možnost podporovat (**Enable Code Folding** – doporučuji zaškrtnout) a které části kódu mají být po otevření souboru skryté (sbalené) – doporučuji žádné, jak je to v implicitním nastavení.



Obrázek 2.12
Zobrazování a skrývání částí kódu

Zobrazení a skrytí částí kódu můžete měnit kdykoliv během editace. Na obrázku 2.12 vidíte výsek programu se dvěma metodami. U první je jak komentář, tak tělo metody zobrazené, u druhé je skryté. Zobrazení či skrytí změníte klepnutím na značku vedle dotyčného kódu, jak to ukazuje kurzorová šipka na obrázku. Kromě toho můžete změnit sbalení všech sekcí daného druhu v celém zdrojovém kódu zadáním příslušného povelu v podnabídce **Code Folds** nabídky **View**.

Karta Formatting

Tato karta umožňuje zadat vaše preference při formátování zdrojového kódu. Popisem možností této karty bych zaplnil klidně dalších 20 stránek, takže opravdu jen ve stručnosti.

Již v základní konfiguraci, kterou jsme instalovali, nabízí rozbalovací seznam **Language** pět možností: společné nastavení **All Languages** a vedle něj pak nastavení pro Javu, JavaScript, JSON a HTML. Nastavení pro Javu je velmi košaté a detailní. Nebudu je zde rozebírat, koho zajímá, může si je proklikat sám a podívat se, co vše se dá pro formátování zdrojových kódů Javy nastavit. Já se tu podrobněji zmíním pouze o společném nastavení, v němž můžete zadat jen zpracování tabulátorů a zalamování řádků.

Určitě bych vám doporučoval zaškrtnout políčko **Expand Tabs to Spaces**, čímž zabezpečíte, že všechny zadané tabulátory budou převedeny na mezery. V opačném případě se totiž může stát, že když otevřete zdrojový kód v jiném editoru, bude zcela jinak zalomen a bude obtížnější se v něm vyznat.

Velikost odsazení po stisku tabulátoru radím ponechat na standardních 4 znacích. Vzdálenost tabulačních zarážek při použití znaků tabulátoru zadávám většinou větší (místo 8 znaků třeba 16), aby se případné zapomenuté tabulátory v cizích kódech co nejmarkantněji projevily a mohl jsem je odstranit.

Nastavení pravého okraje zadávaného v poli **Right Margin**, které specifikuje, kde se bude v kódu naznačovat, že by řádek už měl skončit, doporučuji nechat na standardních 80 znacích. Dlouhé řádky jsou známkou nedbale psaného kódu.

Někdo svoji neochotu řízeně zalamovat řádky maskuje nastavením automatického zalamování řádků po dosažení pravého okraje, které se zadává v seznamu **Line Wrap**. Můžete zadat, že chcete řádek zalomit kdekoliv (**Anywhere**), anebo jen mezi slovy (**After words**). Já však preferuji automatické zalamování vypnuté (**Off**).

Karta Code Completion

Na kartě **Code Completion** zadáváte svá přání, jak by vám měl editor napovídat při tvorbě kódu. S nápovědou aktivovanou stiskem kombinace CTRL+MEZERA jsme se potkali již v *BlueJ*. *NetBeans* ale nabízejí nápovědu na mnohem vyšší úrovni odpovídající požadavkům profesionálních programátorů.

Language

V rozbalovacím seznamu **Language** zadáváte (programovací) jazyk, pro nějž byste chtěli nastavit parametry průběžné nápovědy.

Při zadání **All Languages** nastavujete obecné zadání pro všechny jazyky. Konkrétně zde můžete nastavit:

- **Auto Popup Completion Window** určuje, zda vůbec budete požadovat, aby vám editor při psaní kódu napovídal. Určitě zaškrtnout.
- **Auto Popup Documentation Window** určuje, zda se spolu s nápovědou bude automaticky objevovat i okno s dokumentací nabídnuté třídy, metody či atributu. Doporučuji zaškrtnout.
- **Display Documentation Next to Completion** určuje, zda se bude dokumentace objevovat vedle seznamu nabízených hodnot (zaškrtnuto), nebo někde nad ním či pod ním. Vyberte si podle svého gusta, já nezaškrtávám.
- **Insert Single Proposals Automatically** určuje, zda se bude vůbec něco nabízet v případě, kdy si je editor jistý, jaká hodnota na dané místo patří. Je-li si editor jistý, tak na požádání rovnou doplní potřebnou hodnotu a už vás neobtěžuje dotazy. Mně to vyhovuje, takže zaškrtávám.
- **Case Sensitive Code Completion** určuje, zda bude respektována velikost zadávaných znaků. Mně osobně vyhovuje, když IDE reaguje na písmena, a ne na jejich velikost, takže nezaškrtávám.
- **Show Deprecated Members in Code Completion** určuje, zda budou nabízeny i zavržené (deprecated) třídy a jejich členy. Jako začátečníci byste měli používat pouze perspektivní konstrukce, takže doporučuji nezaškrtávat.

- **Insert Closing Brackets Automatically** určuje, zda se má po zapsání otevírací závorky (uvozovky, apostrofu) automaticky doplnit také zavírací. Mně to vyhovuje, takže zaškrtávám.

Při zadání volby **Java** se objeví specializovanější sada možností:

- **Guess Filled Method Arguments** určuje, že při nabídce doplňování parametrů metod bude editor nabízet proměnné z daného kontextu. Úspěšnost jeho odhadu, která proměnná na dané místo patří, je tak kolem 40 % a záleží na způsobu psaní zdrojového kódu. Když se ale trefí a vy už nemusíte nic psát, je to příjemné.
- **Auto Popup Triggers for Java** určuje, čím můžete automaticky aktivovat nápovědu v průběhu psaní. Implicitně je v poli tečka, takže jakmile napíšete v kódu tečku, editor začne napovídat, co byste za ni mohli napsat.
- **Auto Popup on Typing Any Java Identifier Type** určuje, zda se automatické doplňování bude aktivovat i při zapisování zcela nových identifikátorů. Pak editor většinou nabízí přidat na konec identifikátoru název (případně část názvu) jeho datového typu. Někdy se to hodí, takže nabídku přijmete, jindy vás bude obtěžovat. Zaškrtnutí nechám na vás – já ji mám zaškrtnutou.
- **Completion Selectors for Java** určuje, jaké znaky budou potvrzovat nabídnutou možnost. Zadáte-li libovolný z uvedených znaků, bude editor považovat svoji nabídku za přijatou a nebude identifikátor doplňovat o další vámi psané znaky.
- **Subword completion** umožňuje doplňování i u částí slov. To je novinka *NetBeans* 8.0, na kterou jsem si ještě nezvykl, takže já zatím nezaškrtávám. Zkuste zaexperimentovat, třeba se vám líbit bude.
- Dvoukartový seznam **Package /classes** je další novinkou verze 8.0. Umožňuje zadat balíčky, jejichž obsah se nebude (karta **Exclude**) nebo naopak určitě bude (karta **Include**) zobrazovat v seznamu nabízených identifikátorů. Ve standardní knihovně je totiž řada tříd, které se vyskytují ve více balíčcích. Rozhodnete-li se např. vyvíjet GUI (grafické uživatelské rozhraní) v doporučované knihovně JavaFX, asi nebudete chtít, aby vám editor nabízel stejnojmenné třídy z historického balíčku `java.awt`.

Karta Code Templates

Na této kartě můžete zadat nejrůznější zkratky, které se po zadání rozvinou do části kódu. Např. napíšete-li v kódu `sout` a stisknete tabulátor, napíše se

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

a kurzor se umístí mezi uvozovky. Napíšete-li `psvm` (zkratka z `public static void main`), vloží se do kódu hlavní metoda programu. Takovýchto zkratek je připravena celá záplava a sami si můžete doplnit další.

Karta Hints

Na této kartě si můžete nastavit, na jaké programátorské prohřešky vás má editor upozorňovat. Skoro si říkám, že byste si měli jako začátečníci zaškrtnout všechny, protože byste se měli naučit nedopouštět se žádných programátorských prohřešků. Až budete zkušenější, jistě odhadnete, kdy je výhodnější některou ze zásad porušit, a kdy jde pouze o programátorovu lenost napsat kód správně.

Na druhou stranu ale musím uznat, že řada z hlídaných prohřešků není obecně považována za tak vážné a prosazují je pouze některé skupiny programátorů. Udělal jsem proto malou probírku, a kdo bude importovat moje nastavení, bude s ním importovat i tato doporučení.

Programátorské prohřešky, o nichž jsem hovořil, nejsou syntaktické chyby. Jsou to zlozvyky, které zvyšují pravděpodobnost zanesení chyby do programu. Prostřednictvím těchto narážek či náznaků vás vývojové prostředí jemně upozorňuje, že (v horším případě) porušujete dobré mravy, anebo jenom plně nevyužíváte možností jazyka, které vám umožňují naprogramovat některé obraty elegantněji a/nebo efektivněji.

Karta Highlighting

Na této kartě zaškrtáváte, čí výskyty v kódu chcete zvýraznit v okamžiku, kdy nad danou entitu najedete kurzorem. Osobně doporučuji zaškrtnout vše. Když pak najedete např. nad identifikátor proměnné, zvýrazní se v daném kontextu všechny výskyty dané proměnné. To může být někdy velice užitečné, zvláště při hledání chyb.

Karta Macros

Makra jsou další užitečná pomůcka pro zefektivnění psaní kódu. Makra můžete samozřejmě zadávat i v průběhu psaní v editoru, ale jednou za čas se hodí i možnost upravit přímo jejich kód. Tady tak můžete učinit.

Karta OnSave

Na této kartě můžete nastavit chování *NetBeans* při ukládání souborů. Můžete nastavit zásady platné pro všechny jazyky a pak případně některé speciální zásady pro nějaký konkrétní jazyk. Já preferuji jednotné nastavení pro všechny.

Seznam **Reformat** umožňuje zadat, jestli se má ukládaný soubor ihned automaticky zformátovat. Já se snažím hned při psaní naformátovat programy tak, aby byly co nejpřehlednější, a nestojím o to, aby se mi v tom pak nějaký automat vrtal. Na druhou stranu se ale setkávám se studenty, jejichž kód je tak neskutečně nepřehledný, že bych jim nastavení automatického formátování při ukládání souboru vřele doporučoval.

Seznam **Remove Trailing Whitespaces From** umožňuje zadat, zda se při ukládání budou odstraňovat případné koncové mezery na řádcích. Vhledem k tomu, že tyto mezery jsou v naprosté většině případů opravdu zbytečné, mám nastaveno jejich automatické odstraňování. Zatím jsem se nesetkal se situací, kdy bych potřeboval ponechat nějakou mezeru na konci řádku.

Karta Spellchecker

Tato karta vám umožní zadat vlastní slovník, vůči němuž se bude kontrolovat obsah zadávaných HTML stránek a dokumentačních komentářů. S *NetBeans* se instaluje slovník pro angličtinu, ale budete-li chtít, můžete přidat i slovník pro češtinu či slovenštinu. Já používám slovníky převzaté z editoru PSPad, které můžete stáhnout na adrese http://www.pspad.com/cz/download.php.

2.9 Fonts & Colors – nastavení písma a barev

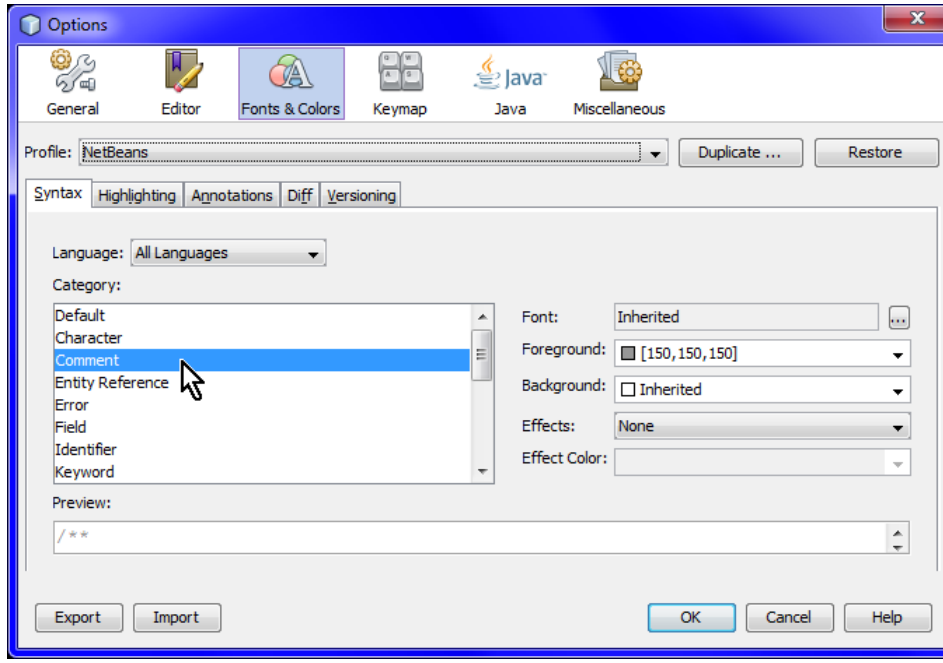
V oblasti **Fonts & Colors** můžete zadat, jaké písmo má používat editor a jakou má mít barvu. Možností je opět celá řada, ale já bych vás chtěl upozornit na dvě: na nastavení optimálního písma a na možnost změny vybarvení komentářů. Obě se nastavují na kartě **Syntax** (viz obrázek 2.13).

Nastavení písma

Pokud vám použité písmo vyhovuje, můžete tuto pasáž přeskočit. Já ale v programech nemám rád písma, u nichž se špatně rozlišuje nula od písmene O a jednička od malého l (L), případně velkého I (i). Ve světě proto vzniklo několik písem vyladěných tak, aby se v nich dobře psaly (a četly) programy.

Používáte-li *Microsoft Windows*, měli byste mít v systému instalované písmo *Consolas*, které *Microsoft* vyvinul speciálně pro programátory. Používáte-li jiný operační systém, můžete instalovat např. písmo *Anonymous Pro*, které si můžete stáhnout na adrese http://www.ms-studio.com/FontSales/anonymouspro.html.

Nastavení společného programátorského písma je jednoduché – vlevo v seznamu **Category** nastavte kategorii **Default** a stiskněte tlačítko vpravo vedle pole **Font**. V následně otevřeném dialogovém okně pak zadejte požadované písmo (přesněji vyberte z instalovaných písem to, které vám bude nejvíce vyhovovat). Nezadáte-li něco jinak, tak všechny ostatní kategorie zadané písmo zdědí.



Obrázek 2.13
Dialogové okno pro nastavování písma a barvy různých textů

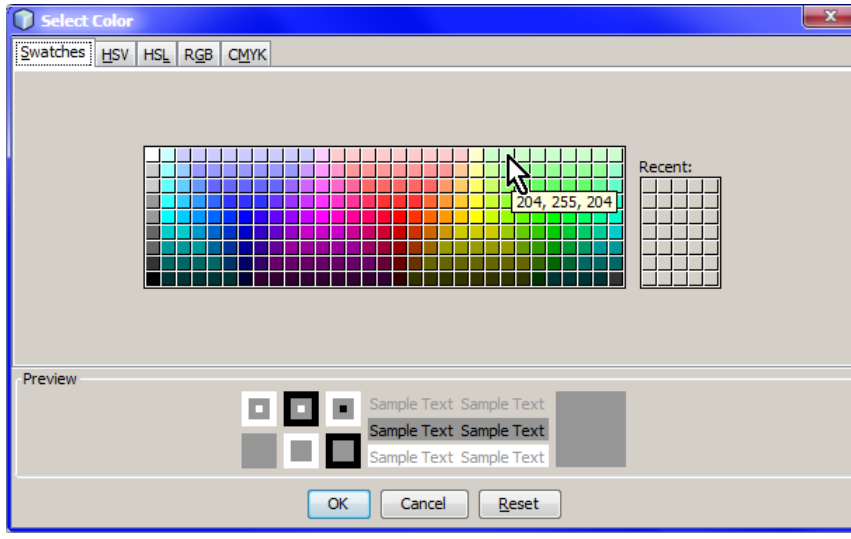
Vybarvení komentářů

V implicitním nastavení mají komentáře světle šedou barvu. Podle mne je toto nastavení nepříjemně nevýrazné, takže když se do nich chce člověk začíst, musí se dost soustředit. Když chce naopak studovat okolní kód, špatně rozlišuje, kde končí komentář a kde začíná kód, protože se od sebe liší jen nepatrně.

Rozhodl jsem se proto, že komentáře ve svých programech odliším barvou pozadí. Protože jsem si v *BlueJ* zvykl na zelené dokumentační komentáře, rozhodl jsem se pro černé písmo na světle zeleném pozadí. Podle mne toto nastavení celý program výrazně zpřehlednilo a mohu vám je vřele doporučit.

Chcete-li si změnit vybarvení kódu podle mého doporučení, postupujte následovně:

1. V seznamu **Category** klepněte na položku **Comment** (viz obrázek 2.13). V pravé části dialogového okna se objeví nastavení písma komentářů. Jak si můžete přečíst, v aktuálním nastavení se písmo (font) i barva pozadí (background) dědí (inherited) od implicitního nastavení. Pro popředí je nastavena šedá barva. Písmo ponecháme, ale změníme barvu pozadí a popředí.
2. Barvu popředí nastavte na implicitní. Klepněte proto na rozbalovací šipku políčka **Foreground** a na konci rozbaleného seznamu klepněte na položku **Inherited**.
3. Barvu pozadí nastavte na naši oblíbenou. (Já ji mám nastavenou na zelenou, ale vy si ji můžete nastavit podle svých preferencí.) Klepněte proto na rozbalovací šipku políčka **Background** a budete-li se stejně jako já domnívat, že nabídnuté barvy jsou pro pozadí příliš výrazné, klepněte na konci seznamu na položku **Custom**.
4. Otevře se standardní okno Javy pro nastavování barvy se základní paletou barev (viz obrázek 2.14). Při ukázání na kteroukoliv z nich se u kurzoru myši objeví velikost jejích RGB komponent. Nebude-li vám vyhovovat žádná barva z nabídnuté palety, klepněte na kartu sady parametrů, jejichž prostřednictvím chcete barvu nastavit, a zadejte požadovanou barvu. Své zadání potvrďte.



Obrázek 2.14
Standardní dialogové okno Javy pro nastavování barvy