4. Balíčky

Obsah

Poznámka:
Tato kapitola není jenom o balíčcích, ale je také hodně o prostředí BlueJ. Výklad je proto trochu podrobnější, než jak danou látku vysvětluji svým žákům. Je to proto, že při práci s balíčky můžete narazit na řadu drobných problémů, které musí vyučující umět řešit. Řadový student by si z ní měl odnést informaci co to jsou balíčky, jak se v prostředí vytvoří nový balíček a jak se zařídí, aby třídy v jednom balíčku mohly používat služeb tříd v jiném balíčku. Informace o chování prostředí BlueJ ve vztahu k balíčkům a o možnosti importu získaných programů jsou určeny spíše pro vyučující.

4.1 Velké programy a jejich problémy

Jakmile začne složitost programu narůstat a tříd začnou být desítky, stovky a tisíce (samotná standardní knihovna Javy má nyní přes 2700 tříd), přestává být únosné udržovat všechny třídy po kupě, protože by se v nich brzy nikdo nevyznal. Je to obdobné, jako když máte všechny papíry na jedné hromadě na stole nebo když máte všechny soubory uložené v jedné složce (znám i taková „čuňata“).

Zkušenost ukázala, že doba, potřebná k vytvoření programu, roste exponenciálně s jeho velikostí. Budete-li schopni vytvořit za jeden den tisíciřádkový program, vůbec z toho nevyplývá, že byste byli schopni vytvořit za týden 5000řádkový program a za měsíc 20000řádkový. Lze spíše očekávat, že k vytvoření 5000řádkového programu budete potřebovat skoro měsíc a 20000řádkový program nevytvoříte o moc rychleji než za rok.

Jedním z cílů správného návrhu je rozdělit program na několik menších částí, které spolu budou pouze minimálně souviset. Každou část je pak možno vyvíjet relativně samostatně a ve vhodných intervalech pouze přezkušovat správnou spolupráci jednotlivých částí.

Jednou takovou částí je třída. Je to však příliš malá část (i když může mít i několik tisíc řádků) a při tvorbě opravdu velkých programů s takovýmto dělením nevystačíme (jak brzy poznáte, nevystačíme s ním dost dobře ani u programů menších). Potřebujeme mít možnost vytvářet části vyššího řádu.

Při správném rozdělení programu můžeme s výhodou využít i to, že uvnitř části si mohou třídy navzájem před sebou odhalit část roušky zapouzdření, aby pak tyto znalosti mohly využít ke zvýšení efektivity svojí práce. Přepokládá se přitom, že všechny třídy uvnitř dané části jsou dílem jednoho nebo několika málo programátorů, kteří jejich kód důvěrně znají a riziko nekorektní modifikace kódu je u nich proto výrazně menší.

4.2 Balíčky

Java umožňuje sdružovat skupiny tříd do tzv. **balíčků** (package). V jednom balíčku jsou přitom třídy, jejichž zdrojové kódy jsou v jedné složce.

> **Zpřesňující poznámka:**
> Ona to není tak docela pravda – správně bych totiž neměl hovořit o zdrojových souborech, ale o přeložených souborech (soubory s příponou .class). Protože však vývojová prostředí dokáží zařídit, aby struktura složek, v nichž jsou umístěny přeložené soubory, kopírovala strukturu složek, v níž jsou umístěny zdrojové soubory, nebudeme se touto drobnou nepřesností dále zatěžovat. Budeme počítat s tím, že umístíme-li správně zdrojový soubor, umístí vývojové prostředí správně i přeložený soubor.

K tomu, aby překladač uznal třídu jako součást balíčku, nestačí pouhé umístění zdrojového textu do příslušné složky. Programátor ještě musí explicitně potvrdit, že soubor sem nezabloudil omylem, ale že sem opravdu patří. Proto je potřeba, aby zdrojový text začínal příkazem:

```
package nazev_balíčku;
```

kde `nazev_balíčku` musí být stejný jako název složky, v níž je soubor umístěn. Protože název balíčku je identifikátorem, logicky z toho vyplývá, že názvy složek s balíčky musí dodržovat pravidla pro tvorbu identifikátorů, tj. smějí obsahovat pouze písmena, číslice a znaky „_“ a „$“, nesmějí začínat číslicí a nesmějí být shodné s žádným klíčovým slovem. Podle konvence by měl navíc název balíčku obsahovat pouze malá písmena.

Poznámka:
Tady musím opět upozornit na problémy s češtinou. Některá prostředí totiž mají problémy s převodem kódů a nejsou schopna správně rozpoznat názvy souborů a složek.

Znovu opakuji, že příkaz **package** musí být prvním příkazem v souboru, před ním smějí předcházet pouze komentáře.

Z úzkého vztahu mezi balíčky a složkami vyplývá, že třída může být pouze v jednom balíčku (nemůžete mít jeden a tentýž zdrojový soubor zároveň ve dvou složkách). Proto také může být na začátku třídy pouze jediný příkaz **package**.

Podbalíčky

Stejně jako mohou složky obsahovat podsložky, mohou balíčky obsahovat podbalíčky. Název podbalíčku vznikne z názvu rodičovského balíčku následovaného tečkou a názvem složky, v níž je umístěn podbalíček.

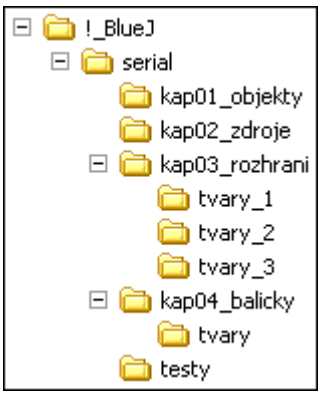
Podbalíčky mohou mít opět podbalíčky, a to libovolné hloubky. Pro název platí stále stejná konvence. Balíčky tak mohou vytvářet stejnou stromovou strukturu jako složky a i pro jejich názvy platí obdobná pravidla, jako pro absolutní cestu k dané složce – pouze se lomítka nebo zpětná lomítka (záleží na operačním systému) nahradí tečkami.

Rozhodnu-li se tedy všechny programy tohoto seriálu umístit do balíčku **`serial`**, ve kterém vyhradím pro každou kapitolu podbalíček, a umístím-li navíc jednotlivé sady postupně vylepšovaného kódu z minulé kapitoly do podbalíčků **tvary_1** až **tvary_3** (viz obrázek 4.1), budou se tyto podbalíčky jmenovat:

```
serial.kap03_rozhrani.tvary_1,
serial.kap03_rozhrani.tvary_2 a
serial.kap03_rozhrani.tvary_3.
```

Zdrojové soubory tříd v prvním z nich by pak měly začínat příkazem:

```
package serial.kap03_rozhrani.tvary_1;
```



Obrázek 4.1: *Stromová struktura balíčků*

Poznámka:
Vykřičníku na počátku názvu kořenové složky nedávejte žádný speciální význam. Má jediný

účel: je prvním písmenem v abecedě, takže zařídí, že se mi daná složka zobrazí v průzkumníku či Total Commanderu v rámci své rodičovské složky mezi prvními.

Na konci seznamu podsložek/podbalíčků složky/balíčku serial na obrázku 4.1 je složka/balíček testy, která nepatří do žádné kapitoly. Prozatím mi slouží pouze k tomu, abych sem umístil programy, pomocí nichž otestuji správnost toho, co jsem pro vás připravil. Časem začneme takto umísťovat balíčky ve chvíli, kdy jejich obsah přiměřeně zkompletujeme. V dalších programech je pak totiž budeme moci používat bez toho, abychom si museli pamatovat kapitolu, ve které jsme je vytvořili.

4.3 Balíčky a BlueJ

Asi se budete ptát, jak počítač pozná, kde má strom balíčků a podbalíčků svůj kořen (na obrázku 4.1 je jím složka !_BlueJ). Odpověď je jednoduchá: sám to nepozná, musíme mu to nějak sdělit.

Každé vývojové prostředí definuje svůj způsob, jak mu oznámíme, ve které složce se tento kořen nachází. BlueJ zachází s balíčky oproti jiným vývojovým prostředím poněkud nestandardně: balíček je po něj takovým malým projektem. Kořenovou složkou stromu balíčků je pak ta složka, jejíž rodičovská složka již neobsahuje soubor bluej.pkg a pro BlueJ proto není projektem.

Poznámka:
Nyní si již můžete sami odvodit, že název souboru bluej.pkg pravděpodobně vznikl ze sousloví *BlueJ package* – v BlueJ totiž můžeme do jisté míry považovat balíček za projekt a projekt za balíček.

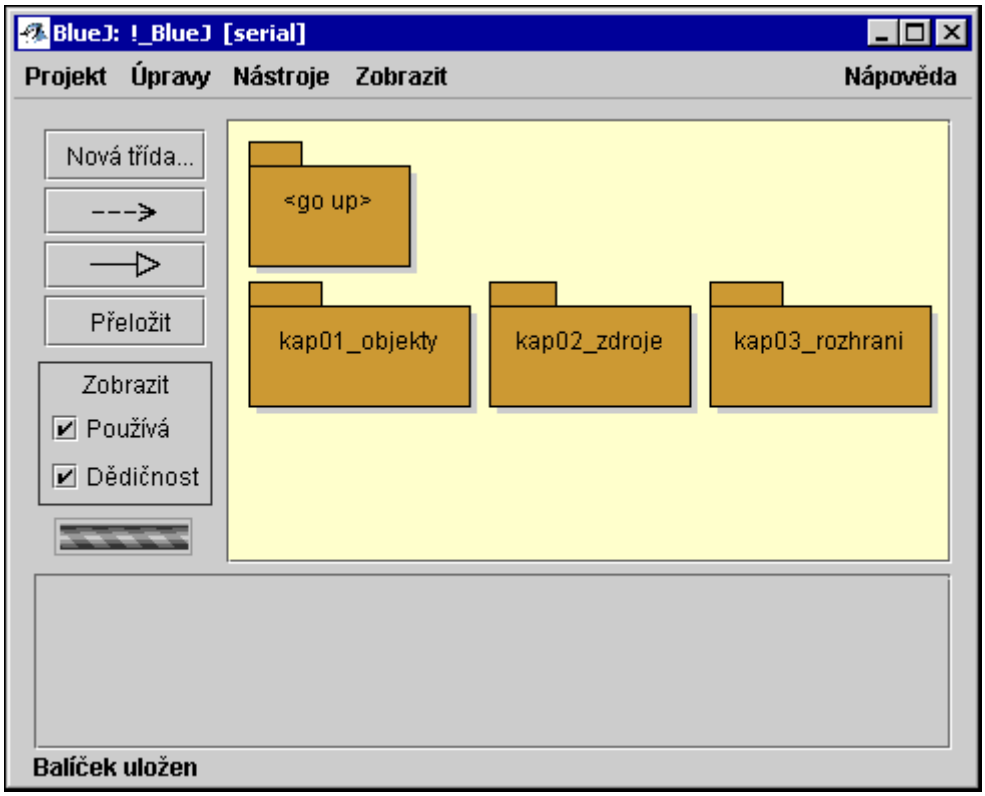
Soubory v připravených složkách

Nejprve si ukážeme, jak bude BlueJ reagovat, připravíme-li mu celou strukturu potřebných složek „ručně“, tj. vyrobíme-li nejenom potřebné složky, ale upravíme-li je rovnou jako projekty (tj. dodáme-li do nich soubory bluej.pkg). Postupujte, prosím, následovně:

1. Ve složce určené pro javové projekty vytvořte podsložku **!_BlueJ** (můžete si ji pojmenovat i jinak, ale na tento název se budu v dalším textu odkazovat), která bude kořenovou složkou našeho budoucího stromu složek/balíčků. Vložte do ní prázdný soubor bluej.pkg
2. Ve složce **!_BlueJ** vytvořte podsložku **serial** a opět do ní vložte prázdný soubor bluej.pkg.
3. Do složky **serial** zkopírujte složky s projekty k prvním dvěma kapitolám a složky přejmenujte podle obrázku 4.1 na **kap01_objekty** a **kap02_zdroje**.
4. Ve složce **serial** vytvořte podsložku **kap03_rozhrani** a vložte do ní prázdný soubor bluej.pkg.
5. Do složky **kap03_rozhraní** přesuňte všechny tři složky s postupně vylepšovanými projekty třetí kapitoly. Tyto složky přejmenujte tak aby výsledek odpovídal příslušné části stromu na obrázku 4.1 (složky pro čtvrtou kapitolu a testy zatím nevytvářejte).
6. Spusťte BlueJ a otevřete projekt **!_BlueJ**, který jste připravili v předchozích bodech.

BlueJ vás asi překvapí tím, že neotevře okno projektu **!_Seriál_BlueJ**, ale správně odhadne, že kořenový balíček je nezajímavý a otevře proto rovnou okno balíčku **serial** (název balíčku, jehož okno je právě otevřeno, je uveden v záhlaví okna v hranatých závorkách za názvem projektu). V okně projektu zobrazí ikony odpovídající jednotlivým balíčkům. Jak si můžete všimnout, balíčky mají svoji vlastní ikonu, která se podobná ikonám složek v správcích souborů.

V levém horním rohu diagramu tříd je ikona balíčku označená **<go up>**. Tato ikona symbolizuje rodičovský balíček a není ji možno nikam přesunout. S ikonami ostatních balíčků můžete hýbat stejně, jako jste mohli hýbat s ikonami tříd. Po přeuspořádání může aplikační okno vypadat např. následovně (připomínám, že jste měli připravit složky pouze pro první tři kapitoly):

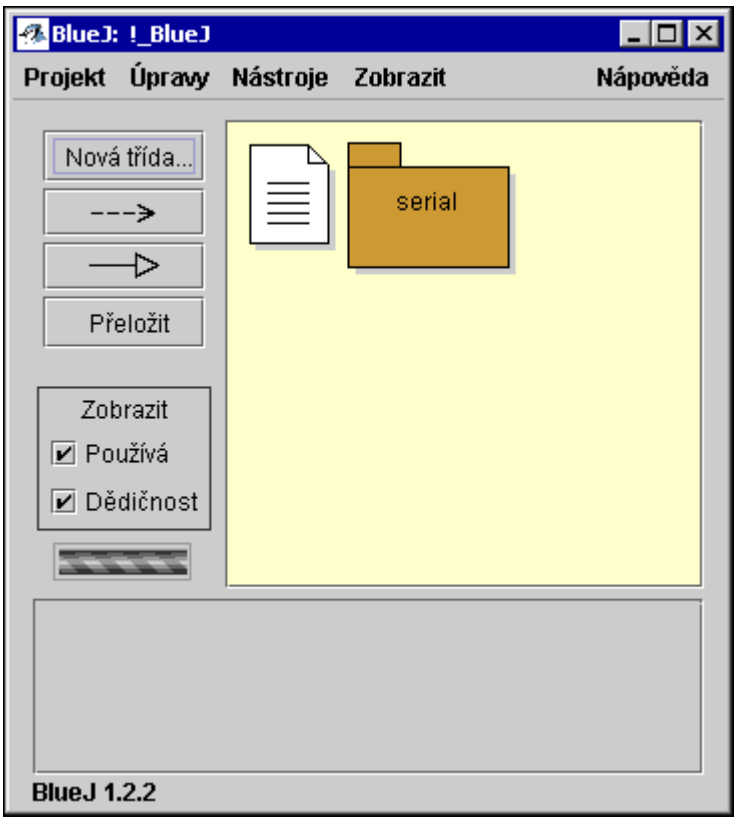


Obrázek 4.2: *Okno balíčku serial projektu !_ BlueJ*

Mezi balíčky se přesouváme poklepáním na jejich ikonu. Jinou možnost bohužel BlueJ nenabízí, takže ti, kterým se chvěje ruka a mají s poklepáním problémy, musí otevření balíčku natrénovat.

Jak jsem řekl, BlueJ se k balíčku do jisté míry chová jako k samostatnému projektu. Chcete-li se proto podívat na obsah jiného balíčku, BlueJ vždy otevře nové okno, abyste mohli mít před očima diagramy tříd obou prohlížených balíčků zároveň.

Poklepejte nyní na ikonu **<go up>**. BlueJ otevře nové okno projektu s diagramem tříd rodičovské složky, obsahující pouze balíček **serial**.



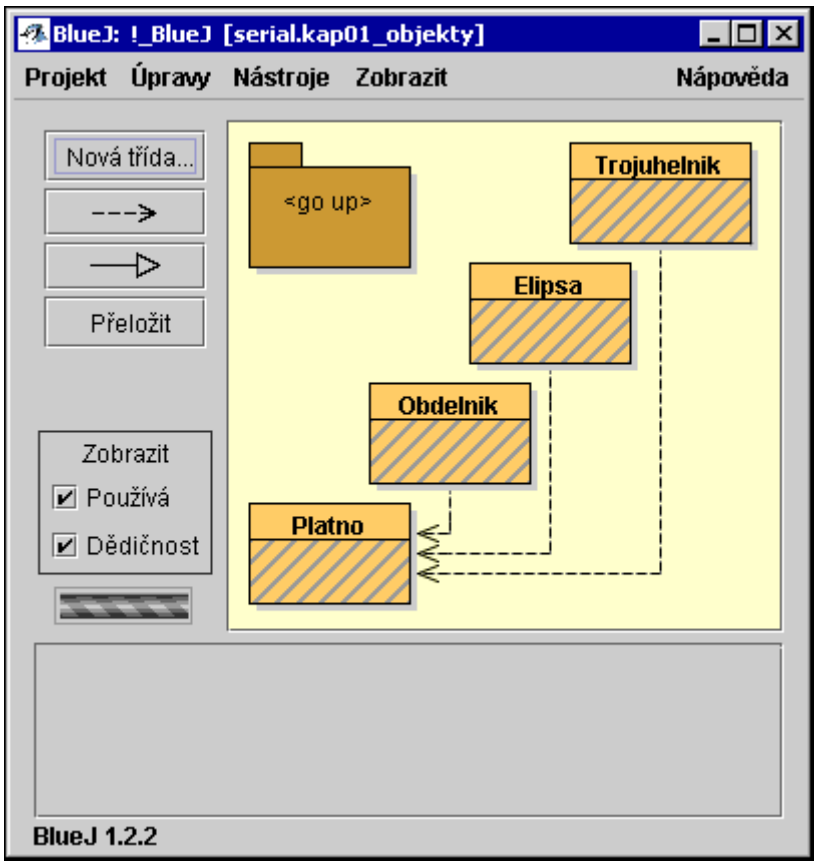
Obrázek 4.3: *Okno kořenové složky stromu balíčků*

Vidíte, že diagram kořenového balíčku je opravdu nezajímavý, takže se poklepáním na ikonu balíčku **serial** přesuneme zpět do jeho před chvílí opuštěného okna. Všimněte si přitom, že si BlueJ pamatuje, že okno cílového balíčku je již otevřeno, takže neotevírá nové okno, ale pouze vás přesune do dříve otevřeného okna.

Okno kořenového balíčku již nebudeme potřebovat a můžem je proto zavřít.

Poznámka:
Při zavírání okna dalšího projektu je jedno, zavřete-li je jako aplikaci (tj. klepnutím na zavírací tlačítko na titulkové liště či stiskem ALT+F4), nebo jako projekt (tj. stiskem CTRL+W nebo zadáním příkazu **Projekt→Zavřít projekt**). Význam těchto dvou příkazů se liší až v případě, kdy je otevřeno jediné okno projektu. Pak je možno zavřít pouze projekt aniž bychom s ním zavírali také okno BlueJ.

Podívejme se nyní na podbalíčky. V okně balíčku **serial** poklepejte na ikonu balíčku **kap01_objekty**. Otevře se okno balíčku **serila.kap01_objekty** s ikonami zkopírovaných tříd. Všimněte si, že v záhlaví okna je v hranatých závorkách opět vypsán celý název balíčku, jehož diagram tříd okno ukazuje. Balíček je zde již uváděn jako podbalíček balíčku **serial**.



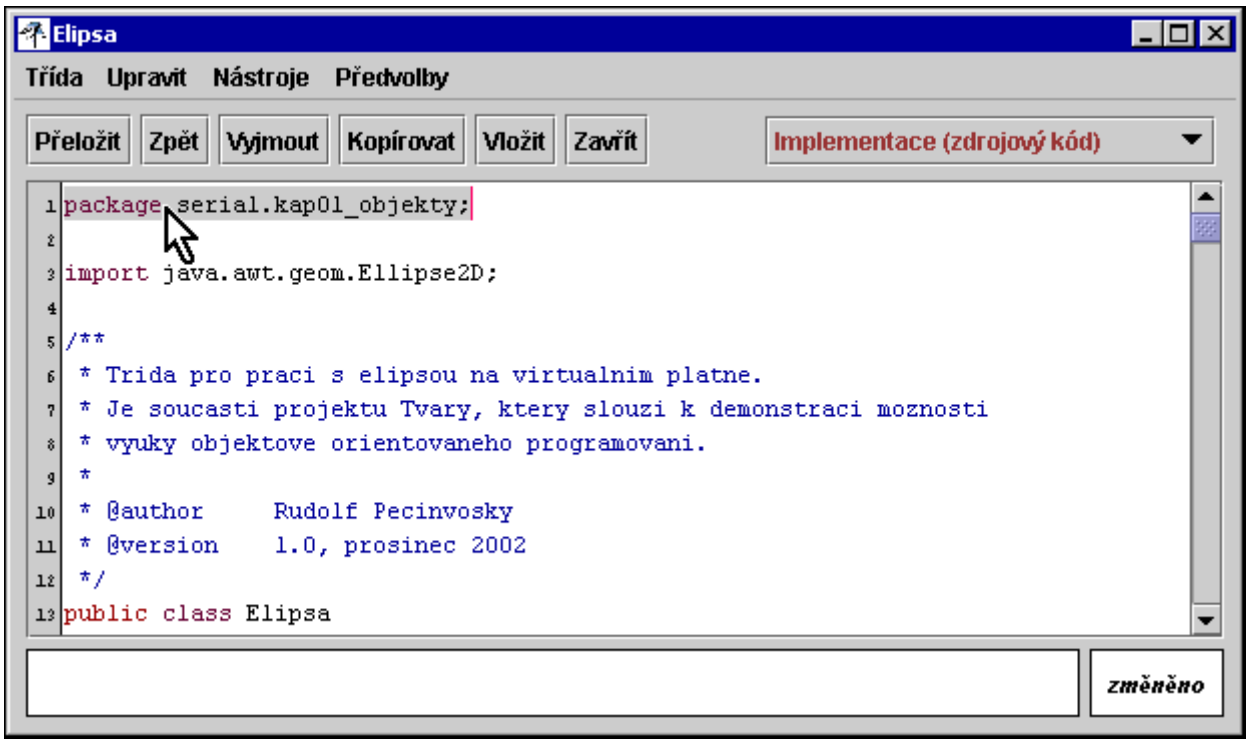
Obrázek 4.4: *Diagram tříd v balíčku serial.kap01_objekty*

Uspořádejte ikony nejprve tak, aby byl diagram přehledný (např. podle obrázku) a otevřete zdrojový soubor třídy **Elipsa**. Na jeho prvním řádku si můžete všimnout, že BlueJ sám iniciativně přidal potřebný příkaz **package**.

Poznámka:
Bohužel i zde narazíme na to, že autoři prostředí proti duchu Javy ignorují jazyky s jinou znakovou sadou. Někdy se stane, že BlueJ soubor správně neidentifikuje jako zdrojový soubor Javy a nejenže na jeho začátek nevloží příkaz **package**, ale často ani není schopen správně natáhnout šipku mezi třídou a rozhraním, které třída implementuje.

Znovu připomínám, že má-li být BlueJ schopen automaticky doplňovat do zdrojových souborů potřebné příkazy **package**, přidávat do zdrojového textu klausule reagující na natažení implementační šipky či vypisovat nápovědu ve spouštěcích oknech metod, nesmí být ve zdrojovém souboru žádné znaky s diakritickými znaménky. Stále platí věta klasika: „Můžeme s tím nesouhlasit, můžeme proti tomu protestovat, ale to je vše, co s tím můžeme dělat.“

Většinou v takovém případě pomáhá důsledně odstranit všechna diakritická znaménka z textu. Znovu bych při této příležitosti doporučil freewarový textový editor PSPad, který tuto možnost nabízí. Já to dělám tak, že pomocí klávesové zkratky CTRL+A vyberu veškerý text, který pak přes schránku vložím do prázdného dokumentu otevřeného v editoru PSPad. Tam jej „odháčkuju“ a opět pomocí CTRL+A přenesu pomocí schránky zpět, kde jím nahradím původní text. Většinou se po této operaci BlueJ vzpamatuje a začne se chovat tak, jak má.



Obrázek 4.5: *Příkaz package vložený na počátek zdrojového textu*

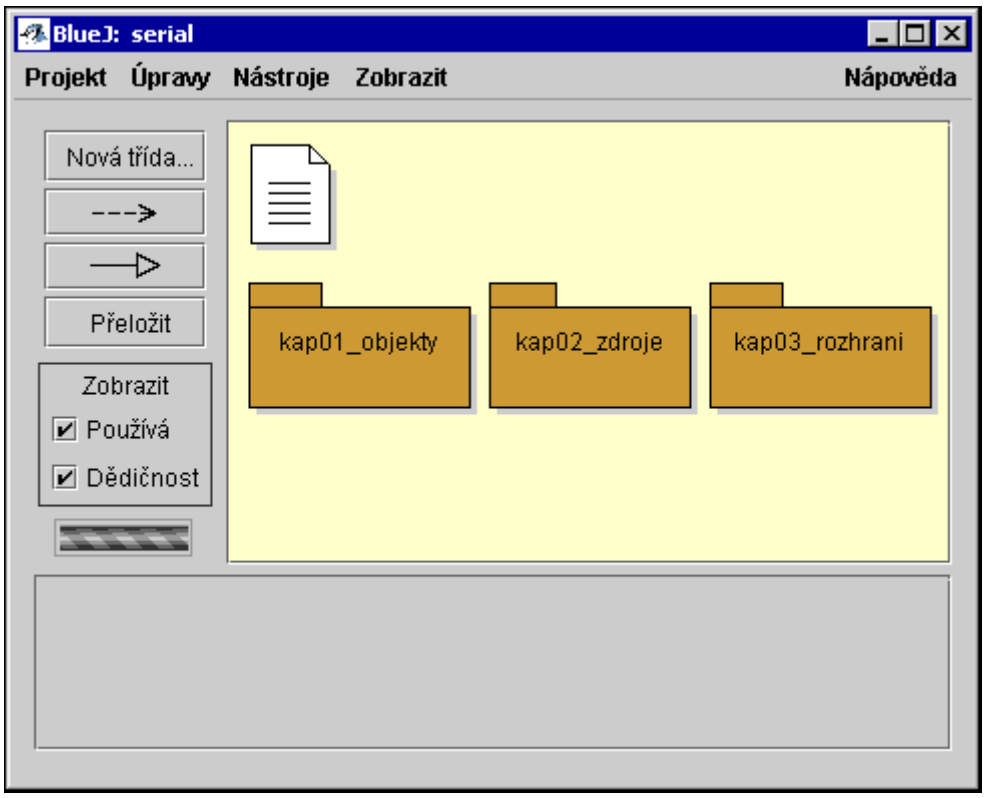
Než půjdeme dále, nechte přeložit třídy v tomto balíčku. Za chvíli se na tento překlad odvolám.

Automatická redefinice balíčků

V předchozím příkladu byly balíčky uspořádány přesně podle požadavků struktury na obrázku 4.1. Podívejme se, co se stane, když nedodržíme to, že složka odpovídající kořenovému balíčku nebude tou nejvrchnější složkou považovatelnou za projekt, tj. obsahující soubor bluej.pkg. Zkusíme z ní tento soubor odebrat a podíváme se, co se stane.

Než ale začnete pracovat s obsahem složky !_BlueJ, zavřete nejprve všechny projekty a poté i celý BlueJ. Pak ve složce smažte soubor bluej.pkg (někdy Windows tento soubor odmítnou smazat, dokud není zavřen BlueJ). Nyní BlueJ znovu spusťte a zkuste zadat příkaz **Projekt → Znovu otevřít → Q:\!_BlueJ** (disková jednotka a případná rodičovská složka u vás může být samozřejmě jiná). Jak se můžete přesvědčit, BlueJ bude váš příkaz ignorovat. Složka !_BlueJ totiž neobsahuje soubor bluej.pkg a není proto balíčkem ani projektem. Tím je až její podsložka serial.

Zadejte tedy příkaz **Projekt → Otevřít** a otevřete jako projekt složku serial. Protože v ní jsme soubor bluej.pkg nemazali, BlueJ ji akceptuje jako složku projektu a zobrazí nám diagram jejich tříd.

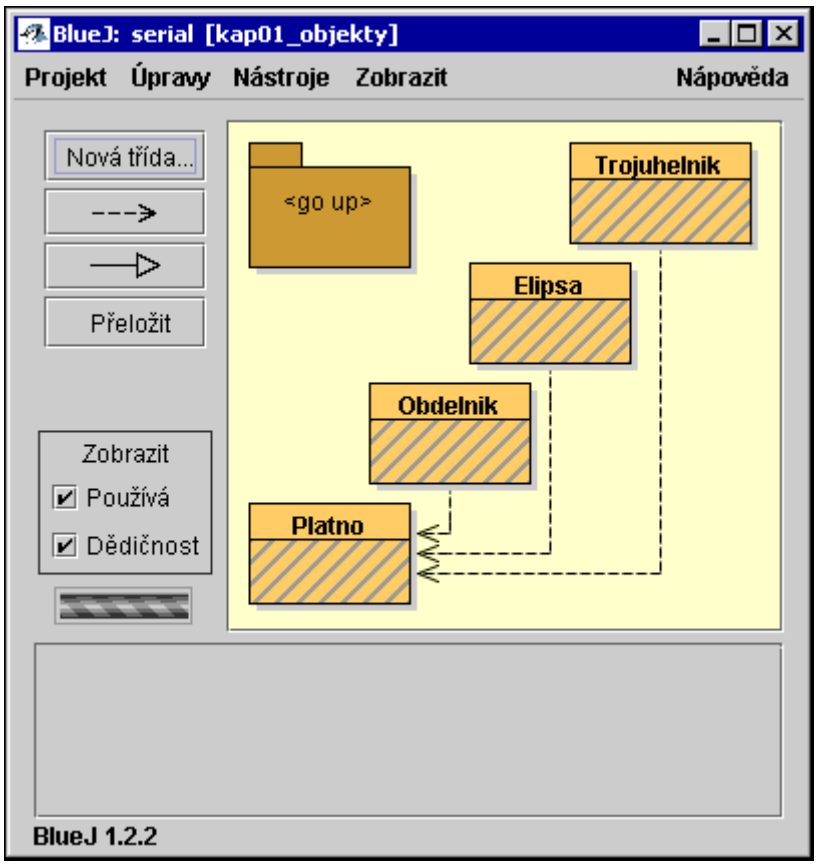


Obrázek 4.6: *Diagram tříd projektu seriál*

V záhlaví okna si všimněte toho, že zde již není v hranatých závorkách uveden název balíčku, protože složka je tentokrát chápána jako kořenová složka projektu.

Poklepeme na ikonu balíčku **kap01_objekty**, který jsme si prohlíželi před chvílí. Opět se otevře nové okno se známým obsahem. V okně si všimněte dvou věcí:

- ☞ V záhlaví okna je balíček **kap01_objekty** uveden jako balíček první úrovně (tj. balíček bez rodiče).
- ☞ Třídy jsou opět nepřeloženy. Jak jistě sami odhadnete, je to proto, že při novém uspořádán bylo třeba v jejich zdrojových kódech změnit příkaz **package** a po této změně již nikdo třídy znovu nepřekládal.



Obrázek 4.7: *Diagram tříd v balíčku kap01_objekty*

Otevřete znovu okno třídy **Elipsa** a překontrolujte, že se příkaz **package** od minula opravdu změnil – nyní definuje třídu jako součást balíčku první úrovně, který již není podbalíčkem balíčku **serial**.

Obrázek 4.8: *Zdrojový kód třídy Elipsa s upraveným příkazem package*

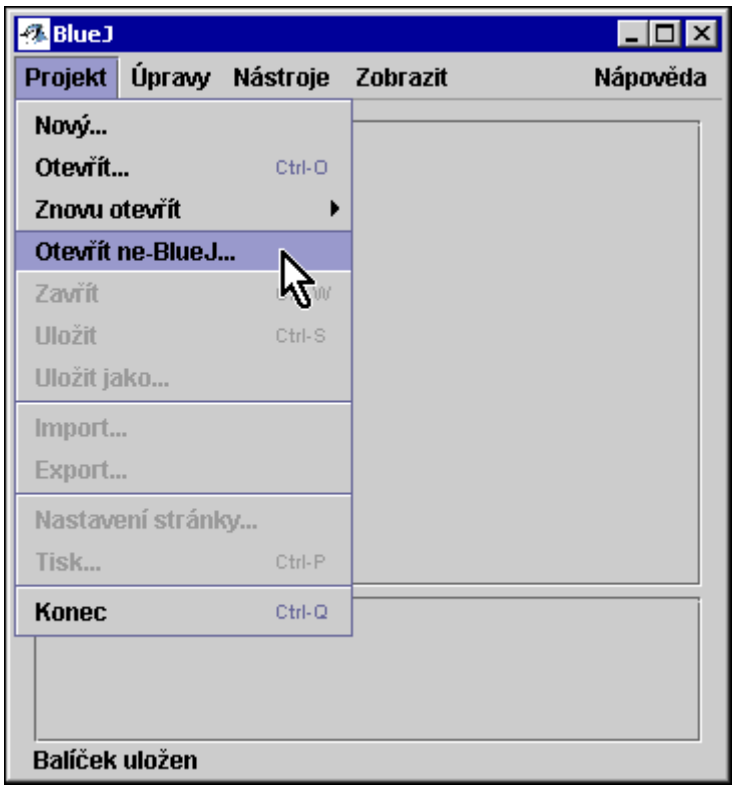
4.4 Práce s projekty vytvořenými v jiných vývojových prostředích

V minulé podkapitole jsme připravovali všechny složky „ručně“, abychom pochopili, jak BlueJ se složkami pracuje a také proto, abychom v budoucnu dokázali odhalit a opravit případné chyby vzniklé chybným smazáním či přidáním souboru bluej.pkg. Prostředí BlueJ nám však umožňuje celý proces zautomatizovat, protože je schopno inteligentně importovat projekt (strukturu balíčků) vytvořenou v jiných prostředích. Stačí mu ukázat na kořenovou složku a BlueJ si do ní a do jejích podsložek doplní všechny potřebné informace samo.

> **Poznámka:**
> Pod pojmem **importovat** rozumím „doplnit strom složek informacemi potřebnými k tomu, aby jej bylo možno akceptovat jako projekt BlueJ“.

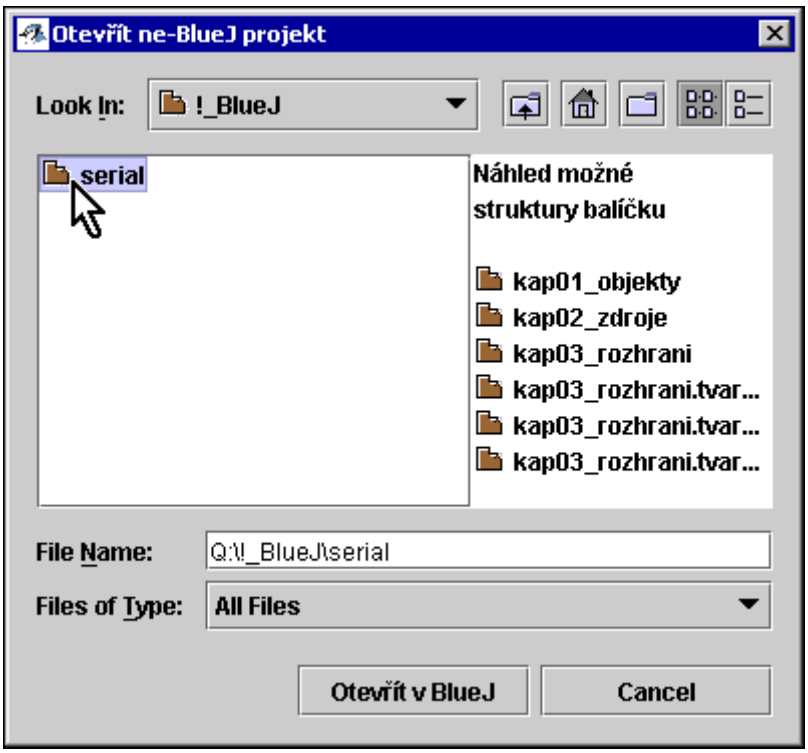
Abychom vše vyzkoušeli, nemusíme naštěstí připravovat uměle nějaký „cizí“ projekt, protože smazáním souboru bluej.pkg ze složky !_BlueJ jsme náš projekt dostatečně „zmršili“, takže je teď pro BlueJ jako cizí. Zavřete proto nyní všechny projekty, abychom si mohli ukázat, jak postupovat v případě, kdy nám někdo věnuje hotový projekt (tj. strom složek-balíčků se zdrojovými soubory), který bychom chtěli dále „oprašovat“ v BlueJ.

K importu projektu, který není BlueJ projektem, slouží příkaz **Projekt → Otevřít ne-BlueJ**.



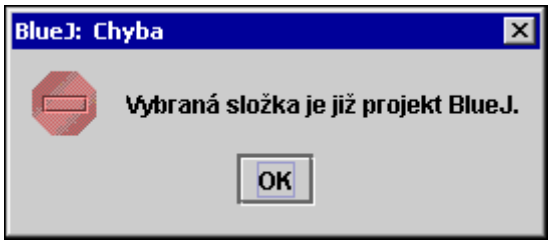
Obrázek 4.9: *Příkaz pro import projektu vytvořeném v jiném vývojovém prostředí*

Zadáním tohoto příkazu otevřeme dialogové okno **Otevřít ne-BlueJ projekt**, které oproti klasickému otevíracímu oknu obsahuje navíc **Náhled možné struktury balíčku**, v němž BlueJ ukazuje, co si myslí o struktuře balíčku, který se chystáme importovat.



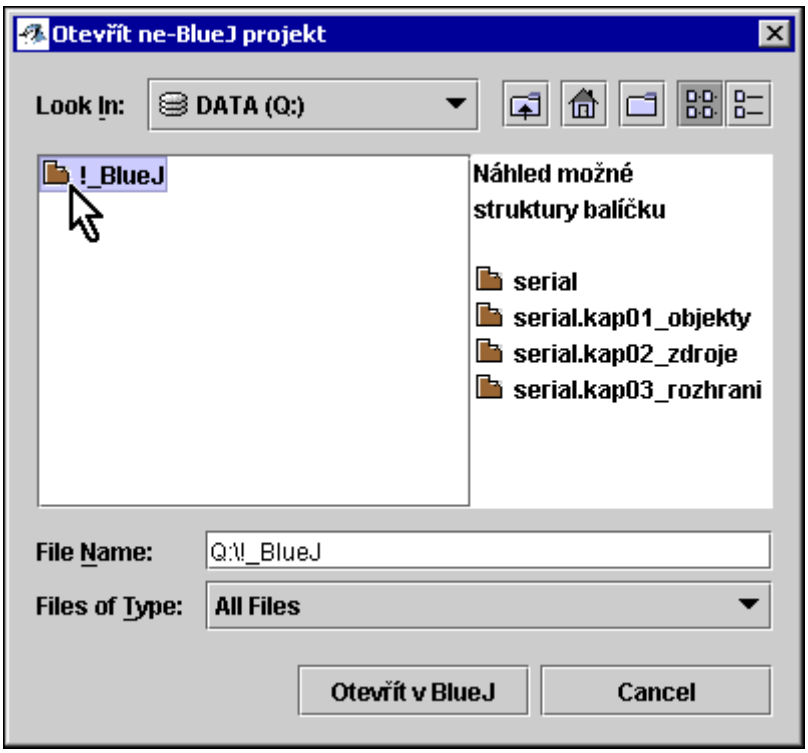
Obrázek 4.10: *Okno Otevřít ne-BlueJ projekt s náhledem možné struktury balíčků v projektu serial*

Pokusme se nejprve importovat náš současný projekt **serial**. „Proklikejte“ se v okně do příslušné složky (viz obrázek) a stiskněte pak tlačítko **Otevřít BlueJ**. Jak vidíte, BlueJ naše zadání neakceptuje a pouze nám oznámí, že zadaná složka je již projektem BlueJ.



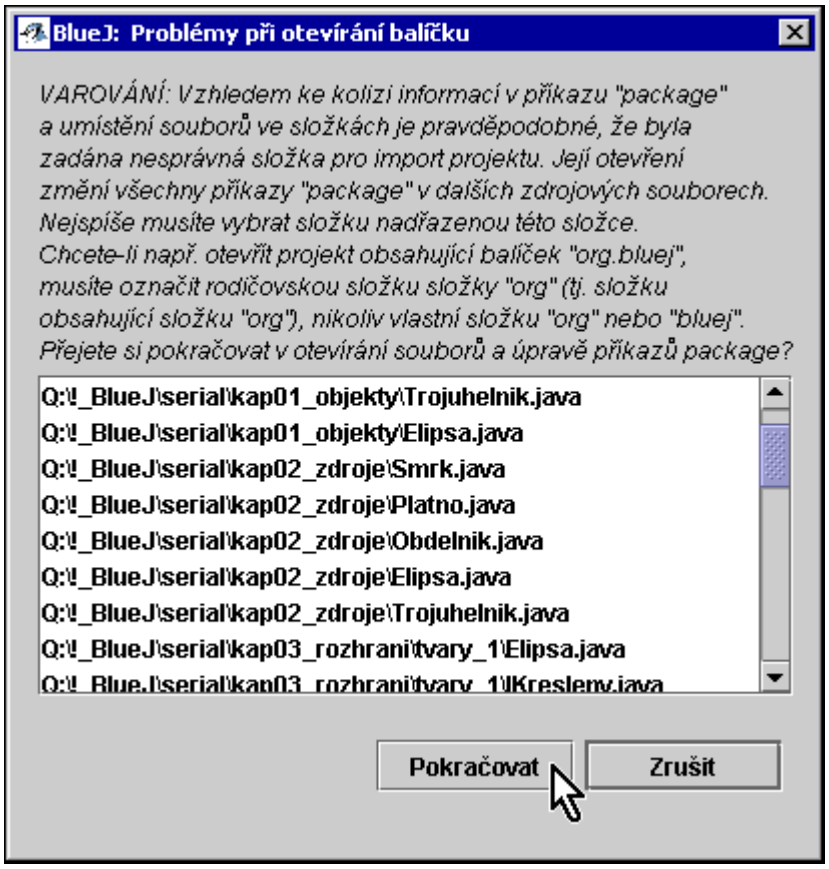
Obrázek 4.11: *Vlastní projekt BlueJ jako cizí neakceptuje*

Otevřeme proto okno znovu a tentokrát najedeme na složku !_BlueJ, o které víme, že BlueJ projektem není, protože jsme z ní klíčový rozpoznávací soubor odstranili.



Obrázek 4.12: *Náhled možné struktury balíčků v projektu !_BlueJ*

Tentokrát již BlueJ naše zadání akceptuje. Při následné kontrole však zjistí, že deklarovaná (tj. ze zdrojového kódu odvozená) struktura balíčků neodpovídá struktuře složek otevíraného projektu, a na tuto skutečnost nás upozorní.



Obrázek 4.13: *Deklarovaná struktura balíčků neodpovídá skutečné struktuře*

Protože víme, že deklarovaná struktura balíčku je výsledkem našeho umělého „poničení“ původního projektu, stiskneme tlačítko **Pokračovat** a necháme BlueJ, ať upraví deklarovanou strukturu balíčků podle skutečnosti.

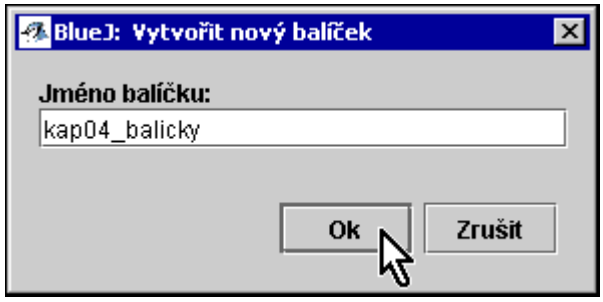
Po skončení operace se můžete po projektu proběhnout a ověřit si, že příkazy **package** jsou opět upraveny tak, aby odpovídaly nové podobě stromu složek/balíčků.

4.5 Vytváření a rušení balíčků přímo v BlueJ

Doposud jsme pro všechny operace s balíčky používali nějakého správce souborů. Nyní si ukážeme, jak lze vytvářet a rušit balíčky přímo v prostředí BlueJ.

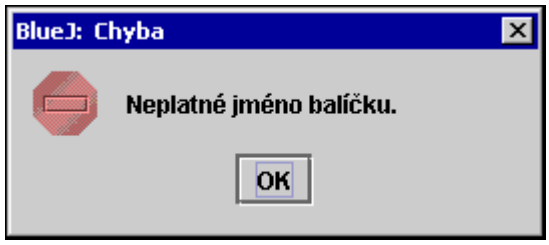
Vytvoření nového balíčku

Otevřete se znovu balíček **serial**. Zadejte nyní příkaz **Úpravy → Nový balíček...**. BlueJ otevře dialogové okno **Vytvořit nový balíček**, v němž po vás chce zadání názvu vytvářeného balíčku. Protože nám stále ještě chybí balíček pro tuto kapitolu, zadáme název **kap04_balicky** a své zadání potvrdíme.



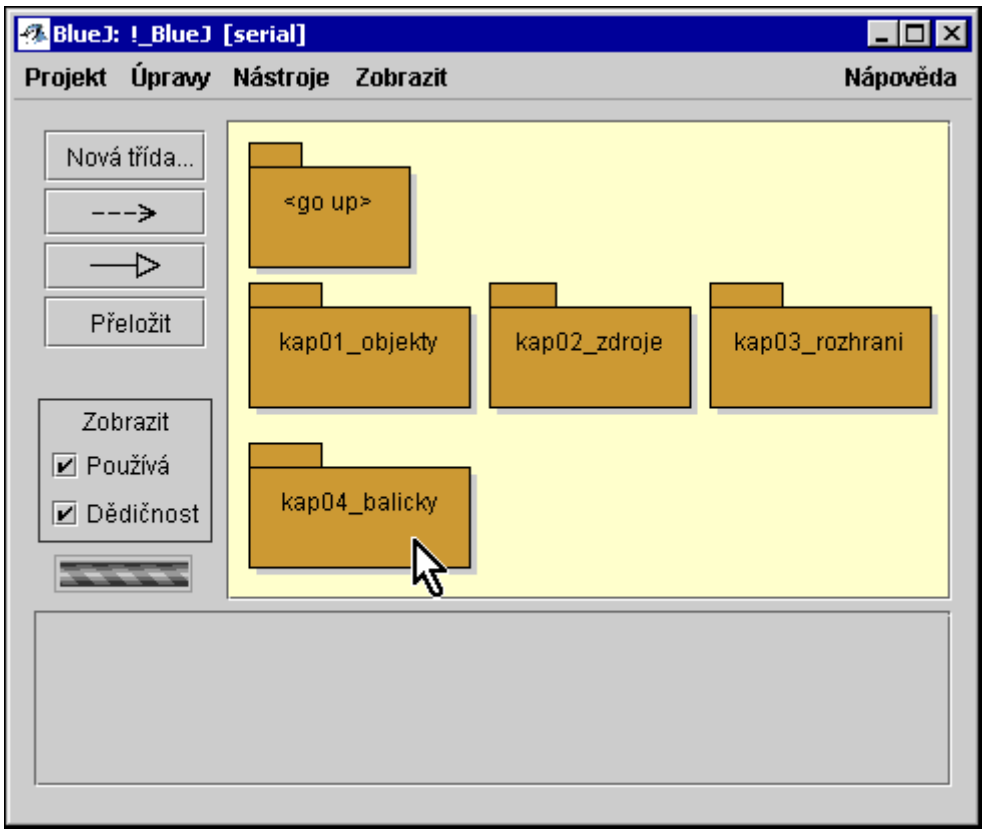
Obrázek 4.14: *Zadání názvu nového balíčku*

BlueJ u zadaného názvu nejprve zkontroluje, zda vyhovuje pravidlům kladeným na identifikátory. Nevyhovuje-li, tak nás na tuto skutečnost upozorní.



Obrázek 4.15: *Upozornění na neakceptovatelný název balíčku*

Bude-li pro něj zadaný název akceptovatelný, vytvoří v aktuální složce/balíčku podsložku/podbalíček (hned v ní vytvoří potřebný soubor bluej.pkg). Nově vytvořený balíček se vzápětí objeví v diagramu tříd.

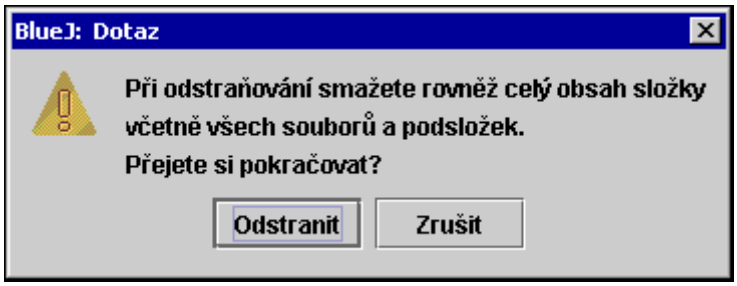


Obrázek 4.16: *Diagram tříd s právě přidaným balíčkem*

Odstranění balíčku

Odstranění balíčku je obdobné odstranění třídy. Pouze nemůžeme využít místní nabídku balíčku (již víme, že balíček žádnou nemá), ale poté, co balíček vybereme, musíme o jeho odstranění požádat příkazem **Úpravy → Odstranit třídu/balíček**.

BlueJ naštěstí na náš příkaz nezareaguje hned, ale nejprve se nás zeptá, myslíme-li to s ním vážně. Přitom nás upozorní na to, že při jeho odstranění zmizí nejen vlastní balíček, ale i všechny třídy a podbalíčky, které s v něm nacházejí.



Obrázek 4.17: *Potvrzovací dotaz před odstraněním balíčku*

4.6 Jmenné prostory a příkaz import

Než si vše, co jsme se o balíčcích dozvěděli, vyzkoušíme při tvorbě vlastního programu, musíme si povědět ještě něco o názvech tříd. V dalším textu budu rozlišovat **vlastní název třídy**, což je název uvedený v hlavičce třídy (např. **Elipsa**) a **úplný název třídy**, který vznikne tak, že vlastní název třídy kvalifikujeme názvem balíčku, v němž se třída nachází (v našem případě např. **serial.kap01_objekty.Elipsa**).

Java povoluje používat vlastní názvy pouze u tříd ze stejného balíčku. Chceme-li pracovat s třídami z jiného balíčku, musíme použít jejich úplný název. Používáme-li však třídy z jiných balíčků často (a to je poměrně běžné), budou nám jejich úplné názvy program nepříjemně znepřehledňovat. Proto autoři jazyka zavedli příkaz **import**, který umožní názvy tříd z označeného balíčku dovézt a používat pak pro ně místo úplných názvu pouze jejich vlastní názvy. Syntaxe příkazu **import** je následující:

Příkaz import

import *název balíčku* **.** *vlastní název třídy*
import *název balíčku* **.** ***

Jak vidíte, příkaz má dvě podoby: první podoba dováží pouze název konkrétní třídy, druhá dováží názvy všech tříd z označeného balíčku.

Přimlouvám se za to, abyste dali přednost první podobě. Dovážíte-li názvy několika tříd z jednoho balíčku, musíte sice uvést pro každou z nich vlastní příkaz **import**, ale na druhou stranu bude na první pohled zřejmé, které třídy z cizích balíčků používáte.

> **POZOR!**
> I u názvů balíčků platí, že **Java rozlišuje velikost znaků**. Pracujete-li ve Windows, tak víte, že tento operační systém velikost znaků v názvech souborů a složek sice zobrazí, ale jinak ji nebere na vědomí. Složka **serial** a **SERIAL** je pro něj jedna a ta samá složka. Ne tak pro Javu. V příkazu **import** proto musíte použít přesně stejnou velikost znaků, jakou mají názvy příslušných souborů a složek.
>
> Prozatím za nás potřebné soubory vytvářel BlueJ. Nyní jsme si potřebné složky vytvořili sami a sami pak na ně budeme v příkazech import definovat odvolávky. Musíme proto dbát na to, abychom je nedefinovali tak nešikovně, že by si pak s nimi překladač nevěděl rady.

Podbalíčky

Při práci s balíčky musíme mít neustále na paměti, že **podbalíček není součástí balíčku**. Budu-li proto někdy tvrdit, že cosi platí pro všechny třídy v balíčku, bude to platit opravdu jenom pro ně a nesmíte si dané tvrzení přenášet na třídy v podbalíčcích.

Budete-li chtít proto používat třídy v podbalíčku, musíte jejich názvy importovat stejně jako názvy tříd z kteréhokoliv jiného balíčku.

Importujete-li proto pomocí hvězdičkové konvence všechny názvy tříd z daného balíčku, neimportujete tím ještě názvy tříd z jeho podbalíčků. Budete-li je chtít používat, musíte je importovat zvlášť. Znovu ale opakuji, že používání hvězdičkového tvaru se považuje za méně čisté

zvlášť. Znovu ale opakuji, že používání hvězdičkového tvaru se považuje za méně čisté a některá prostředí dokonce vydávají v takovém případě varovné zprávy.

Balíček java.lang

Zvláštní postavení mezi všemi balíčky má balíček **java.lang**. V tomto balíčku jsou základní třídy, které jsou používané ve většině programů. Proto je tento balíček dovezen vždy implicitně, takže se již nemusí explicitně dovážet prostřednictvím příkazu **import**.

My jsme prozatím používali z tohoto balíčku jedinou třídu, a to třídu **java.lang.String**. Brzy však dojde i na další.

4.7 Příklad – testovací program

Vyzkoušejme si, co jsme se právě naučili. Definujme v balíčku **serial.kap04_balicky** třídu **Test** s veřejnou statickou metodou **test**, která otestuje, že všechny třídy v podbalíčku **tvary** dělají to, co slibují, a dělají to správně.

> **Poznámka:**
> Některé z vás již možná unavuje spouštět po každé opravě programu metodu **AnimPlatno.getPlatno()** a vytvářet pak jednotlivé objekty, abyste na nich mohli vyzkoušet, zda je váš program bez chyb. Nyní si ukážeme, jak lze takovéto testy částečně automatizovat: vytvoříme třídu, v níž naprogramujeme činnosti, prostřednictvím nichž vše po každé opravě programu vždy otestujeme.

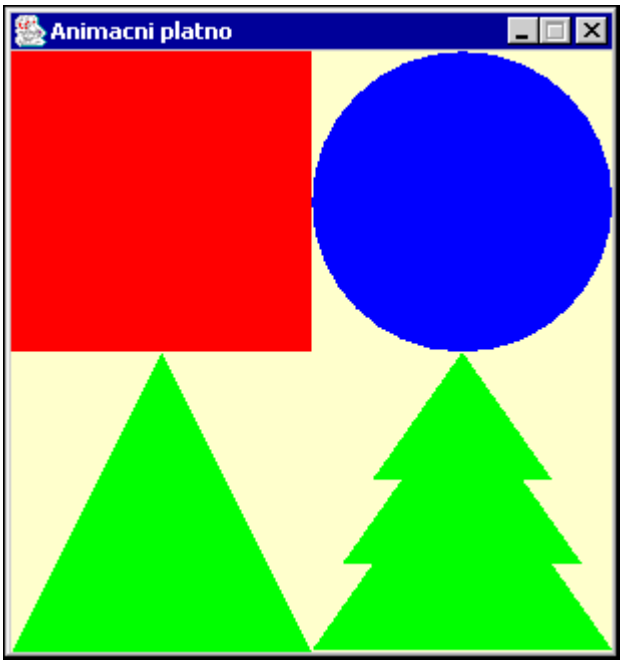
Po metodě **test** budeme chtít, aby v levém horním rohu plátna vytvořila 4 různé grafické tvary (obdélník, elipsu, trojúhelník a smrk), přesunula je do levých horních rohů jednotlivých kvadrantů plátna a zde je roztáhla přes celou plochu příslušného kvadrantu. Všechny třídy, které bude testovat, budou přitom umístěny v podbalíčku **serial.kap04_balicky.tvary**.

Bez dokumentačních komentářů by definice této třídy mohla vypadat např. následovně:

```
package serial.kap04_balicky;

import serial.kap04_balicky.tvary.AnimPlatno;
import serial.kap04_balicky.tvary.Elipsa;
import serial.kap04_balicky.tvary.Obdelnik;
import serial.kap04_balicky.tvary.Trojuhelnik;
import serial.kap04_balicky.tvary.Smrk;
import serial.kap04_balicky.tvary.Presouvac;
import serial.kap04_balicky.tvary.Kompresor;

//Predchozi prikazy importu je mozno nahradit jedinym prikazem:
//import serial.kap04_balicky.tvary.*;

/**
 * Trida Test slouzi k otestovani spravne funkce trid v podbalicku tvary.
 *
 * @author     Rudolf PECINOVSKY
 * @version    1.00, 5.3.2003
 */
```

```
public class Test
{
    public static void test()
    {
        //Pripravim zakladni objekty
        AnimPlatno  ap = AnimPlatno.getPlatno();
        Obdelnik    o = new Obdelnik();
        Elipsa      e = new Elipsa();
        Trojuhelnik t = new Trojuhelnik();
        Smrk        s = new Smrk();

        //Casto pouzivane konstanty
        int s2 = ap.getSirka()/2;
        int v2 = ap.getVyska()/2;

        //Preda vytvorene komponenty do spravy platna
        ap.pridej(o);
        ap.pridej(e);
        ap.pridej(t);
        ap.pridej(s);

        //Test presouvace
        Presouvac p = new Presouvac(2);
        p.presunNa(e, s2,  0 );
        p.presunNa(t,  0, v2 );
        p.presunNa(s, s2, v2 );

        //Test kompresoru
        Kompresor k = new Kompresor(2);
        k.nafoukniNa(o,  s2, v2 );
        k.nafoukniNa(e,  s2, v2 );
        k.nafoukniNa(s,  s2, v2 );
        k.nafoukniNa(t,  s2, v2 );
    }// public static void test()

}//public class Test
```

Po skončení běhu metody **test()** bud plátno vypadat následovně:



Obrázek 4.18: *Výsledná podoba animačního plátna po testu*

Úkol:
Definujte metodu **test2()**, která zavolá metodu **test()**, poté zdvojnásobí rozměry plátna, smaže jeho obsah a znovu spustí metodu **test()**.

Povinné testy

Od této chvíle budeme v každé nově definované třídě definovat také statickou metodu **test**, která nám při svém zavolání otestuje, že třída pracuje přesně tak, jak bylo naším cílem.

V současné době je stále populárnější metodika *extrémního programování*, která dokonce tvrdí, že programátor má nejprve napsat testy a teprve pak vlastní program. Tento postup má dvě výhody:

- ☞ Při psaní testů si ujasníte, co má vlastně vytvářený program dělat.
- ☞ V průběhu tvorby programu si můžete kdykoliv ověřit nakolik se blížíte k vytouženému cíli.

Extrémní programátoři dokonce vyvinuli speciální program, který takovéto testování maximálně zjednodušuje a automatizuje. Až budou naše znalosti dostatečné k tomu, abyste mohli tento program začít používat, tak vás s ním seznámím.

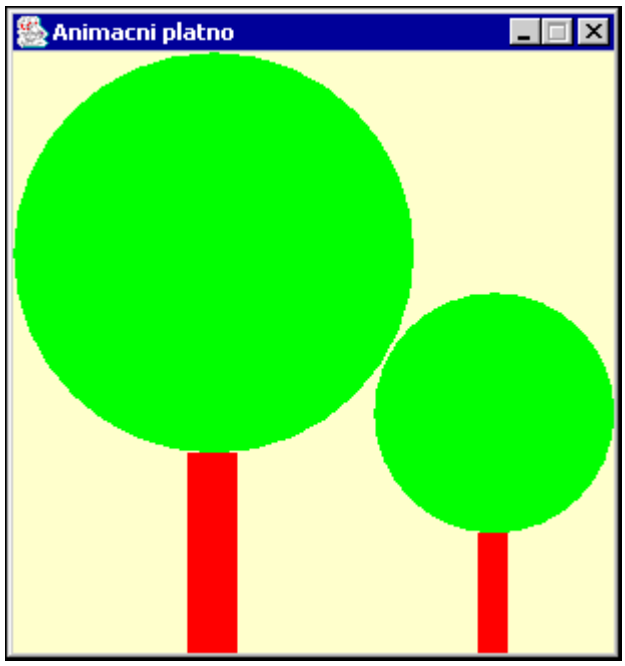
4.8 Používání tříd a rozhraní z jiného balíčku

Používání balíčků je velice výhodné. Od této chvíle již budu všechny další programy umísťovat do balíčků, které budou postupně rozšiřovat strom z obrázku 4.1.

Zavedení balíčků má však pro uživatele BlueJ jednu nevýhodu: přijdou o možnost definovat implementaci rozhraní třídou prostým natažením šipky. BlueJ totiž neumí natáhnout šipku z balíčku do balíčku. Budeme-li proto potřebovat implementovat rozhraní z jiného balíčku, nezbude nám, než napsat příslušnou klauzuli do hlavičky třídy ručně.

Při specifikaci implementovaného rozhraní máme (obdobně jako při použití jakýchkoliv jiných identifikátorů) dvě možnosti: buďto uvedeme v hlavičce jeho plný název nebo vložíme za příkaz **package** odpovídající příkaz **import** a v hlavičce pak použijeme pouze vlastní název daného rozhraní.

Úkol:
Rozšiřte množinu zobrazitelných tvarů o třídu **Jablon**, která vznikne spojením zelené elipsy (koruna) o zadané šířce a výšce rovné dvěma třetinám zadané výšky stromu a červeného obdélníku o (kmen) o výšce rovné třetině zadané výšky a šířce rovné 1/8 zadané šířky. Tuto třídu definujte v balíčku **serial.kap04_balicky**. Třídu definujte tak, aby implementovala všechna rozhraní definovaná v balíčku **serial.kap04_balicky.tvary**. Výsledná nakreslená jabloň by mohla vypadat např. následovně:



Obrázek 4.19: *Obrázky dvou instancí třídy Jablon*

Se zdrojovým textem třídy **Jablon** tu nebudu zaclánět – jak si jistě domyslíte, bude svým stylem velice podobný zdrojovému textu třídy **Smrk** nebo vašich tříd **Snehulak** a **Domek**. Jediným principiálním rozdílem bude sada příkazů **import** na počátku souboru (nebo jeden příkaz využívající hvězdičkovou konvenci). Druhým rozdílem bude statická metoda **test()**, kterou jsme dříve nevytvářeli a která by mohla vypadat např. následovně:

```
public static void test()
{
    Jablon j = new Jablon( 0, 0, 100, 150);
    ap.pridej(j);
    Presouvac p = new Presouvac();
    p.presunNa(j, 100, 100);
    Kompresor k = new Kompresor();
    k.nafoukniNa(j, 200, 300);
    Jablon j2 = new Jablon( 180, 120, 120, 180 );
    ap.pridej(j2);
    p.presunNa( j, 0, 0 );
}//public static void test()
```

Poznámka:
Celý strom balíčků/složek z obrázku 4.1, v němž jsme se v této kapitole pohybovali, najdete v projektu **04_Balíčky** na adrese http://vyuka.pecinovsky.cz/.

V tomto souboru najdete kompletní stromovou strukturu složek se zdrojovými soubory příslušných tříd a potřebnými soubory bluej.pkg. Aby nebyl zbytečně velký, nebude obsahovat přeložené třídy – v případě zájmu si je tedy budete muset nejprve přeložit.

V dalším textu již budou soubory k příštím kapitolám obsahovat pouze zkomprimované balíčky, které budou určeny k přidání do této struktury.

BlueJ a vytváření instancí tříd z jiných balíčků

Budete-li chtít vyzkoušet správnost definice třídy **Jablon** přímo tak, jak jste byli donedávna zvyklí, dostanete se do problémů. Autorům vývojového prostředí BlueJ se totiž nepodařilo dotáhnout práci s balíčky do konce. Jednou z funkcí, které tomuto prostředí chybějí, je možnost vytvoření instance třídy z jiného balíčku. Jakmile totiž klepneme na třídu z jiného balíčku, aktivuje se celé okno daného balíčku a po zadání příslušného příkazu z místní nabídky se odkaz na vytvořenou instanci umístí v zásobníku odkazů projektu odpovídajícímu balíčku, v němž třída, jejíž instanci jsme vytvořili, leží.

Tuto neschopnost BlueJ lze obejít např. tak, že si v balíčku vytvoříme třídu, která bude obsahovat sadu statických metod, které dokáží příslušné instance tříd z jiných balíčků vytvořit nebo alespoň požádat příslušné třídy o vrácení odkazu na jejich instance. Vracené odkazy těchto přemosťovacích metod se pak budou umísťovat do zásobníku odkazů toho balíčku, v němž bude definována ona přemosťovací třída.

V balíčku **serial.kap04_balicky** je pro tento účel definována třída **MostTvary**. Otevřete-li její místní nabídku, uvidíte seznam statických metod určených k získání odkazů na instance tříd z balíčku **serial.kap04_balicky.tvary**.

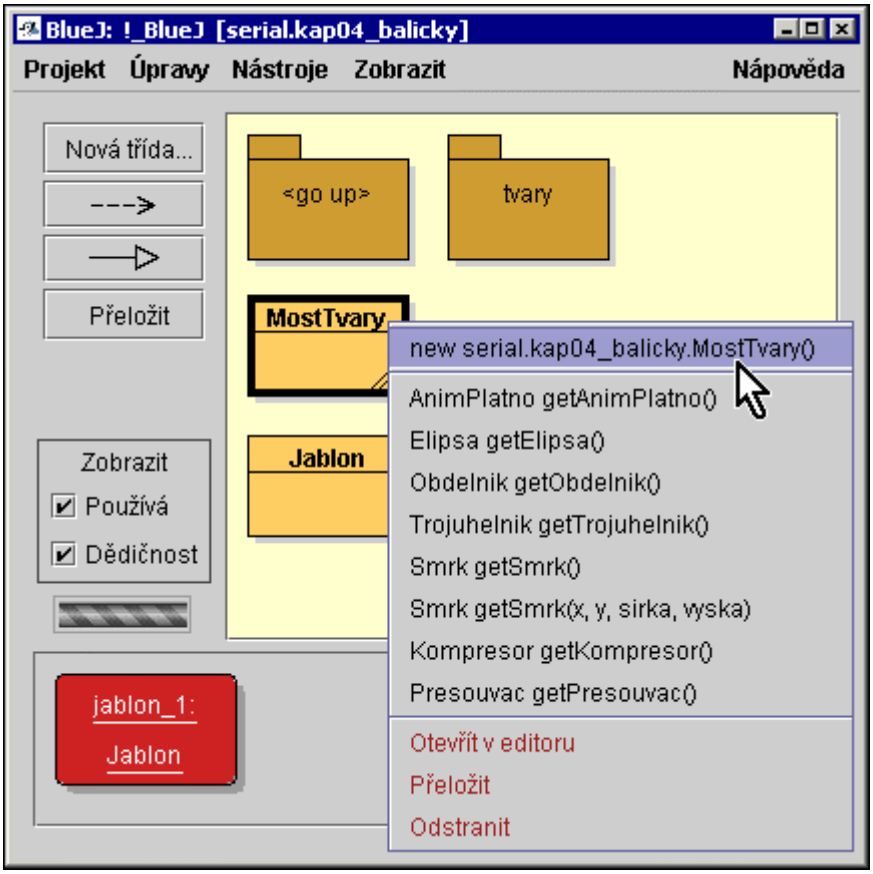
Tato třída slouží pouze jako kontejner na sadu statických metod, pomoci nichž získáváme odkazy na instance tříd z balíčku tvary. Vytvoření její instance proto nemá žádny smysl a je vhodné mu zabránit. Dosáhneme toho vytvořením soukromého konstruktoru. Tento konstruktor bude pro jiné třídy nepřístupný a svou existenci zabraní překladači, aby vytvořil veřejně dostupný implicitní konstruktor (ten překladač vytvoří pouze tehdy, nedefinujeme-li vůbec žádný konstruktor). Její zdrojový kód (bez dokumentačních komentářů) je následující:

```
public class MostTvary
{
    public static AnimPlatno  getAnimPlatno() { return AnimPlatno.getPlatno(); }
    public static Elipsa      getElipsa()     { return new Elipsa(); }
    public static Obdelnik    getObdelnik()   { return new Obdelnik(); }
    public static Trojuhelnik getTrojuhelnik(){ return new Trojuhelnik(); }
    public static Smrk        getSmrk()       { return new Smrk(); }
    public static Kompresor   getKompresor()  { return new Kompresor(); }
    public static Presouvac   getPresouvac()  { return new Presouvac(); }

    public Elipsa getElipsa( int x, int y, int w, int h, String c ) {
        return new Elipsa( x, y, w, h, c );
    }//public Elipsa getElipsa( int x, int y, int w, int h, String c )

    public Obdelnik getObdelnik( int x, int y, int w, int h, String c ){
        return new Obdelnik( x, y, w, h, c );
    }//public Obdelnik getObdelnik( int x, int y, int w, int h, String c )

    public Trojuhelnik getTrojuhelnik( int x, int y, int w, int h, String c ) {
        return new Trojuhelnik( x, y, w, h, c );
    }//public Trojuhelnik getTrojuhelnik( int x, int y, int w, int h, String c )

    public static Smrk getSmrk( int x, int y, int sirka, int vyska ) {
        return new Smrk( x, y, sirka, vyska );
    }//public static Smrk getSmrk( int x, int y, int sirka, int vyska )

    public Kompresor getKompresor( int sila ) {
        return new Kompresor( sila );
    }//public Kompresor getKompresor( int sila )
```

```
    public Presouvac getPresouvac( int rychlost ) {
        return new Presouvac( rychlost );
    }//public Presouvac getPresouvac( int rychlost )

    //Soukromy konstruktor blokujici vytvoreni instaci.
    private MostTvary() {}

}//public class MostTvary
```

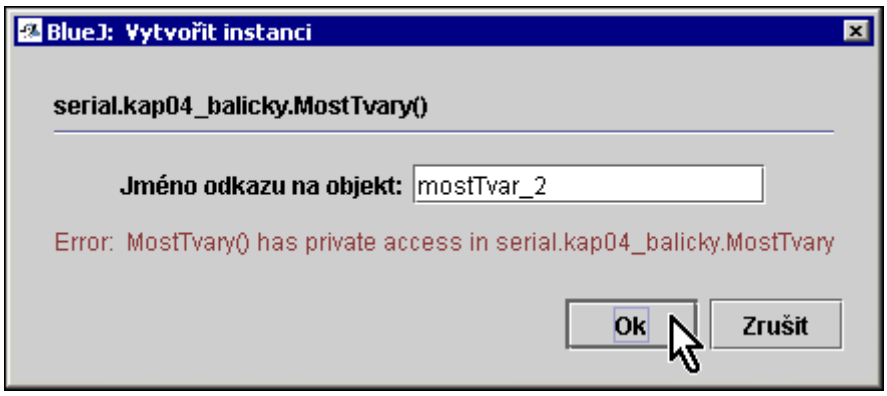
Ukazuji vám je proto, abyste si mohli v případě potřeby vytvořit obdobné vlastní přemosťující třídy.

Upozornění:
Znovu opakuji, že nutnost vytvářet „mostní třídy“ je rys specifický pro prostředí BlueJ, které oproti ostatním vývojovým prostředím nabízí některé výhodné funkce (např. přímé vytváření instancí), ale tyto funkce nedokáže vyvolat přes hranice balíčků.



Obrázek 4.20: *Místní nabídka třídy* ***MostTvary*** *a nepřístupným konstruktorem*

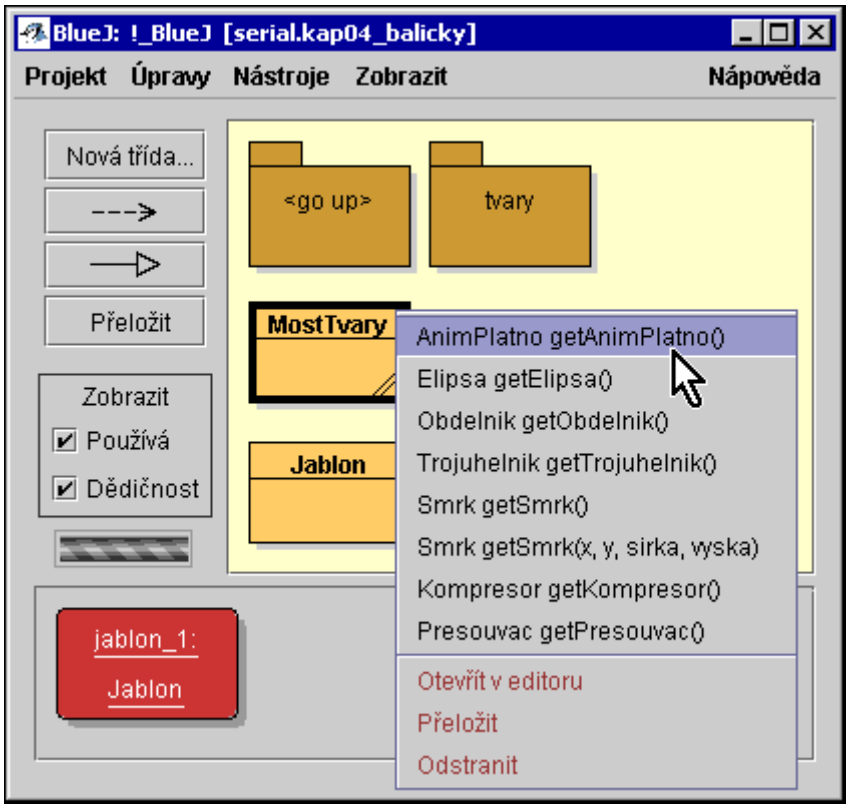
Poznámka:
Před chvílí jsem říkal, že jsme konstruktor znepřístupnili tím, že jsem jej definovali jako soukromý. Možná jste si ale na obrázku si všimli, že v místní nabídce třídy **Most** je přesto konstruktor uveden. To se vás však BlueJ pouze snaží nachytat. Rozhodnete-li se totiž konstruktor zavolat, BlueJ vám oznámí, že máte smůlu, protože konstruktor je definován jako soukromý (*…has private access…*).



Obrázek 4.21: *Soukromý konstruktor není možné zavolat*

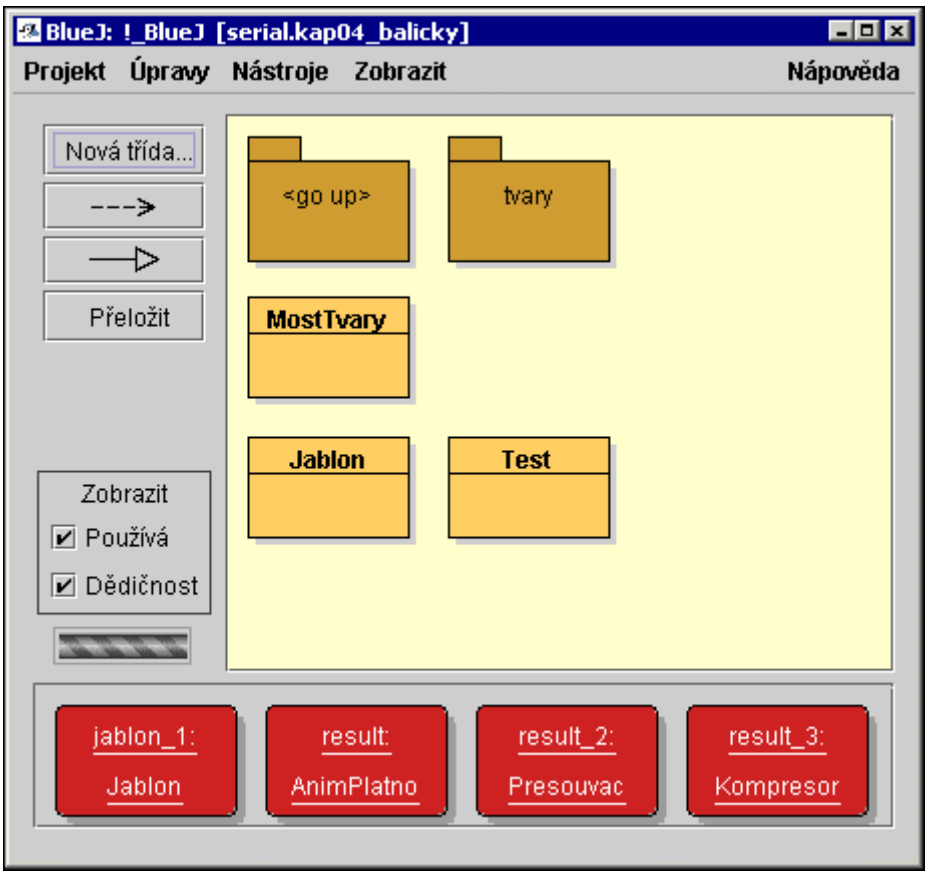
Poznámka – pokračování:
Nepřístupný konstruktor BlueJ zobrazí jen někdy. Nepodařilo se mi však vybádat, co jej k tomu vede. Po nějaké době uvědomil, že třídy **Platno** a **AnimPlatno** měly také soukromý konstruktor, avšak ten se v místní nabídce neukazoval. Chtěl jsem se podívat, v čem se definice obou tříd liší, ale asi jsem BlueJ polekal, protože najednou přestal soukromý konstruktor zobrazovat i u třídy **MostTvary**. Rozpovídal jsem se tu o tom proto, že jsem chtěl, abyste se nepolekali, pokud by někdy příště udělal BlueJ něco podobného i vám.

Budete-li mít to štěstí, že na vás BlueJ nebude „vymýšlet“ žádné podrazy, měla by místní nabídka třídy **MostTvary** vypadat následovně:



Obrázek 4.22: *Správná podoba místní nabídky třídy MostTvary*

Zavolejte postupně metody **getAnimPlatno()**, **getPresouvac()** a **getKompresor()** a nechte vrácené odkazy umístit do zásobníku odkazů. Pak už vám nebude nic bránit v tom, abyste umístili čerstvě vytvořenou jabloň na plátno a vyzkoušeli, že jí lze přesouvat i nafukovat.



Obrázek 4.23: *Vytvořené instance tříd z balíčku* ***serial.kap04_balicky.tvary***

4.9 Přístupová práva v rámci balíčku

Výhodou umístění všech souvisejících tříd do jednoho balíčku není pouze přehlednost a snadnější distribuce. Třídy, které jsou spolu uvnitř jednoho balíčku mají totiž vůči sobě větší práva, než třídy, které jsou každá v jiném balíčku. Můžeme to chápat tak, že třídy v balíčku spolu kamarádí a proto si vzájemně i více důvěřují. Jsou proto ochotny umožnit svým „kamarádům“ přístup k některým složkám (atributům a metodám), k nimž třídy z jiných balíčků nepustí.

Složky, k nimž je třída ochotna pustit své „kamarády“ nemají žádný modifikátor přístupu (prozatím jsme se setkali s modifikátory **private** pro soukromé složky a **public** pro veřejné složky). Pro jejich označení se používá bud výstižnější termín *package private* (soukromé v balíčku) a nebo méně výstižný (avšak v testu lépe použitelný) termín *friendly* – přátelské. Tímto termínem je budu v dalším textu označovat i já (hlavně proto, že se lépe skloňuje ☺).

Použití přátelských složek usnadňuje tvorbu programů. Umožňuje totiž třídám sdílet v rámci balíčku některé pomocné metody a proměnné, o kterých nemusí ostatní třídy vůbec vědět. Této možnosti byste však neměli zneužívat. Každé takovéto sdílení totiž nabourává zapouzdření jednotlivých tříd a přináší tak riziko, že při některé z budoucích úprav zapomenete na nějakou důležitou závislost a zanesete do programu chybu.

V naší knihovně tvarů by např. nebylo nejmoudřejší, kdyby třída **AnimPlatno** zpřístupnila ostatním třídám svůj atribut, v němž uchovává odkaz na používané plátno. Třídy by sice nemuseli pokaždé o tento odkaz žádat, avšak při jakékoliv budoucí úpravě třídy **AnimPlatno** bychom se vy-

stavovali riziku, že změnu chování tohoto atributu zapomeneme přenést do všech tříd, které jej používají.

Dohodněme se proto, že ne-konstantní atributy budeme dále deklarovat jako soukromé a jako přátelské budeme deklarovat pouze pomocné konstanty a metody, které mají své omezené použití pouze v rámci svého balíčku.

4.10 Jedinečnost názvů balíčků

Říkali jsem si, že k zavedení balíčků vedly hlavně dva důvody:

- ☞ Snaha zapouzdřit třídu před budoucími „zákazníky“ (tj. třídami, které ji budou používat) a naopak snaha odkrýt trochu tohoto zapouzdření před svými kolegy.
- ☞ Minimalizace počtu názvů objektů, které třída zná a na které se může obracet. V opačném případě hrozí, že u opravdu rozsáhlých projektů si programátoři nedokáží všechny dostupné názvy pamatovat a ve svých programech pak některé názvy zamění.

V době, kdy si programátoři vyměňují své programy po celém světě by se mohlo zdát obtížné zabezpečit, aby žádní dva programátoři nedefinovali balíček (a v něm i třídu) se stejným názvem. Konvence, které se pro tvorbu těchto názvů zavedli, však podobnou možnost úspěšně vylučují (přesněji vylučují ji pro ty, kdo je dodržují).

Podle těchto konvencí se název balíčku vytvoří tak, že se nejprve uvede obráceně psaná internetová adresa webu jeho tvůrce (obráceně psaná znamená, že se názvy domén uvádějí v obráceném pořadí) a teprve za ní pak následuje název, který pro balíček vymyslel jeho tvůrce. Mám-li tedy internetovou adresu **pecinovsky.cz** (počáteční **www** nebo **vyuka** jsou již názvy složek na příslušném internetovém serveru), měly by názvy všech mých balíčků, u nichž mi záleží na tom, aby jejich názvy nekolidovaly s názvem jakéhokoliv jiného balíčku na světě, začínat **cz.pecinovsky**. Balíček, v němž jsou programu pro tuto kapitolu, by se proto měl správně jmenovat např. **cz.pecinovsky.serial.kap04_balicky**.

Tato konvence má své výhody i nevýhody. Výhodou je to, že podle ní vytvořené názvy jsou opravdu jedinečné, protože na jedinečnost internetové adresy dohlíží internetové servery a na jedinečnost v rámci počítače pak dohlédne operační systém. Nevýhodou této konvence je, že takto vzniklé názvy balíčků jsou velmi dlouhé a v textech obsahujících úplné názvy tříd (programy, chybová hlášení, instalační soubory apod.) se pak čtenář poměrně špatně orientuje.

Ustálená praxe je proto nyní taková, že programy určené k distribuci mezi zákazníky nebo obecně zveřejňované na internetu tuto konvenci dodržují, avšak programy lokálního dosahu (např. doprovodné texty k učebnicím) ji příliš nedodržují a dávají přednost stručnějším názvům, se kterými se lépe pracuje.

Balíčky doprovodných knihoven

Přiznám se, že volba správných názvů byla pro mne velké dilema. U příkladů vztahujících se přímo k učebnici jsem měl jasno – dal jsem přednost kratšímu názvu balíčku. Nemohl jsem se však dlouho

rozhodnout, kterou cestou se vydat při tvorbě názvů knihoven, s nimiž začneme pracovat od příští kapitoly. Krátké verze se mi nelíbily, protože mi pravděpodobnost jejich zopakování připadala nezanedbatelně veliká a správně utvořené názvy byly přeci jenom příliš dlouhé. (Vyzkoušel jsem je a chybová hlášení se stal téměř nečitelná.)

Nakonec jsem se rozhodl pro kompromis, který sice nedodržuje přesně konvence, ale který pravděpodobnost kolize názvů rozumně snižuje a přitom název balíčku příliš neprodlužuje: rozhodl jsem se definovat „kořenový“ balíček **rup** a všechny knihovní balíčky umístit do něj.

Od příští kapitoly občas zveřejním vedle adres souborů se zdrojovými texty programů i adresu některé z doprovodných knihoven. Jejich balíčky budou dodržovat moji výše popsanou upravenou konvenci.

4.11 Co jsme se v kapitole naučili

V této kapitole jsme se seznámili s balíčky, jejich funkcí a použitím. Nyní byste již měli vědět, že:

- Do jednoho balíčku patří třídy, jejichž přeložené soubory jsou ve stejné složce.
- Balíčky tvoří hierarchickou strukturu, která odpovídá struktuře složek.
- Každé vývojové prostředí definuje vlastní způsob jak označit kořenovou složku stromu balíčků.
- Název balíčku sestává z vlastního názvu balíčku, kterému předchází název rodičovského balíčku oddělený tečkou.
- Vlastní název balíčku je shodný s názvem složky obsahující třídy patřící do tohoto balíčku.
- Příslušnost třídy k balíčku musíme definovat v příkazu **package**.
- Příkaz **package** musí být úplně prvním příkazem v souboru před nímž smějí předcházet pouze komentáře.
- Název třídy je dán názvem balíčku následovaným tečkou a vlastním názvem třídy.
- Chceme-li pracovat s třídami z jiných balíčků a nechceme-li používat jejich úplné názvy, musíme jejich názvy nejprve dovézt pomocí příkazu **import**.
- Před příkazem **import** smí předcházet pouze příkaz **package** nebo jiný příkaz **import**.
- Podbalíček nepatří do balíčku.
- Dovezením všech názvů tříd v daném balíčku ještě nedovážíme třídy jeho podbalíčků – ty potřebují vlastní příkaz **import**.
- Jediný balíček, který není třeba importovat, je systémový balíček **java.lang**.
- BlueJ se k balíčkům chová jako k samostatným projektům.
- Pro BlueJ je kořenovým balíčkem balíček, který již nemá rodičovskou složku, jež by mohla být prohlášena za BlueJ projekt.

- ☞ Implementace rozhraní z jiného balíčku není v BlueJ možno definovat natažením příslušné šipky, ale musí se zadat do zdrojového kódu „ručně“.
- ☞ V projektu není možno přímo vytvořit instance tříd z jiných balíčků. Tyto instance lze vytvořit např. pomocí speciální přemosťovací třídy se sadou statických metod zprostředkujících volání příslušných konstruktorů a statických metod.
- ☞ Součástí každé definice třídy bude od nynějška metoda, která otestuje správnou činnost třídy.
- ☞ Při tvorbě programů bývá výhodné napsat nejprve test a teprve pak testovaný program.
- ☞ Neuvedeme-li u metody či atributu žádný modifikátor přístupu, bude tato metoda (atribut) považována za soukromou v rámci balíčku, tj. budou k ní mít přístup pouze třídy z daného balíčku.
- ☞ Uvolnění přístupu k atributům vede k nebezpečnému snížení zapouzdření a měli bychom je proto používat je v opravdu odůvodněných případech.
- ☞ Chceme-li, aby názvy našich tříd byly jedinečné na celém světě, umístíme je do balíčku, jehož název začíná naší obráceně psanou internetovou adresou (názvy domén vyšší úrovně se píší před názvy domén nižší úrovně).

Autor pracuje jako EDU expert ve firmě Amaio Technologies, Inc.