Kapitola 1
Co byste měli znát z prvního dílu

Co se v kapitole naučíte

V této kapitole si velmi stručně připomeneme, co jsme se naučili v prvním dílu učebnice. Ti, kteří četli první díl, si tu pravděpodobně nic nového nenaučí. Tato část je určena hlavně těm, kteří otevírají tuto knihu bez toho, že by četli první díl. Praxe totiž ukázala, že pro leckteré zkušené programátory jsou mnohé z dále uvedených informací novinkou.

Jednotlivé podkapitoly následujícího přehledu odpovídají stejnojmenným kapitolám prvního dílu.

1.1 Seznamujeme se s nástroji

- Program je předpis zapsaný v nějakém programovacím jazyce a definující, jak má procesor, pro nějž je určen, splnit zadanou úlohu.
- Programy dělíme na překládané, interpretované a hybridní. Někdy se obdobné dělení používá i pro programovací jazyky.
- Hybridní programy spojují výhody obou. Překládají se do mezijazyka optimalizovaného pro rychlou interpretaci. Ten je pak interpretován programem označovaným jako virtuální stroj.
- Java je moderní programovací jazyk, který je jednoduchý, objektově orientovaný a jeho programy jsou přenositelné mezi platformami.
- Javu zařazujeme mezi hybridní jazyky. Programy napsané v Javě se překládají do bajtkódu, který je pak interpretován virtuálním strojem.
- Java je nejenom jazyk, ale také platforma, přesněji čtyři platformy: *Java SE* (Standard Edition), *Java EE* (Enterprise Edition), *Java ME* (Micro Edition) a *Java Card*. Pro výuku se ve vstupních kurzech používá *Java SE*.
- Pro vývoj programů v Javě potřebujeme instalovat programovou sadu nazývanou JDK.
- Při vývoji programů se většinou používají speciální vývojová prostředí označovaná zkratkou IDE (*Integrated Development Environment*).
- Současná vývojová prostředí organizují vyvíjené programy do tzv. projektů.
- Při různých vlastností programu používáme grafický jazyk UML, který definuje několik druhů diagramů. Architekturu programu většinou znázorňujeme v *diagramu tříd*.
- Třídy jsou v diagramu tříd znázorněny rozdělenými obdélníky, v jejichž horní části je název třídy.

1.2 Objekty a třídy

- Každý program je jakousi simulací reálného nebo virtuálního světa.
- OOP chápe okolní svět jako svět objektů, které sdružujeme do tříd.
- Objekty patřící do nějaké třídy označujeme jako *instance* dané třídy. Termíny objekt a instance můžeme většinou používat jako synonyma.
- Třídy jsou zvláštní druhy objektů:
  - V Javě jsou to jediné objekty, které nejsou instancí žádné třídy.
  - Jsou to jediné objekty, které mohou vytvářet jiné objekty – své instance.
  - Java nám neumožňuje pracovat s objekty tříd přímo, ale vždy pouze prostřednictvím jejich zástupců.
- Definice třídy obsahuje dva druhy definic svých členů:
  - definice instančních členů, které jsou vlastnictvím jejích instancí, a
  - definice třídních členů, které jsou vlastnictvím celé třídy.
- Třídní členy bývají většinou označovány jako *statické*, instanční členy bývají občas označovány jako nestatické.
- Objekty mezi sebou komunikují prostřednictvím zpráv.
- Objektově orientovaný program je v nějakém programovacím jazyce zapsaný popis objektů (tříd a jejich instancí) a zpráv, které si tyto objekty mezi sebou posílají.
- Reakci na zaslanou zprávu má na starosti speciální část programu označovaná jako *metoda*.
- *Konstruktory* jsou speciální metody zodpovědné za správnou inicializaci nově vytvořených instancí svých tříd.
- V jazyku Java vytváříme nové objekty zasláním zprávy s klíčovým slovem `new` následovaným názvem třídy, jejíž instanci chceme vytvořit, a seznamem parametrů v kulatých závorkách.
- Systém reaguje na zaslání zprávy ve dvou fázích:
  - v první fázi se vyhradí potřebná paměť (to má na starosti virtuální stroj),
  - v druhé fázi se zavolá konstruktor, kterému se předá odkaz na vyhrazenou paměť spolu s dalšími případnými parametry a který vyhrazenou paměť inicializuje.
- Po ukončení inicializace konstruktor předá volajícímu programu odkaz na právě vytvořený objekt.
- Neposkytne-li třída konstruktor, nemůžeme vytvářet její instance.
- O vyhrazení paměťového místa pro vytvářený objekt a o jeho úklid poté, co přestaneme objekt potřebovat, se stará správce paměti.
- Správce paměti bývá v anglických textech označován jako *garbage collector*, což v překladu znamená sběrač odpadků, popelář či uklízeč.
- Program v Javě nikdy nepracuje s objektem, ale vždy pouze s odkazem na objekt.
- Objekt se stane kandidátem na odstranění v okamžiku, kdy už na něj už nikdo neodkazuje, tj. když už jej nikdo nepotřebuje.
- O tom, jestli a kdy má být příslušný objekt doopravdy zrušen, rozhoduje správce paměti.
- Pojmenovaná místa v paměti, do nichž můžeme ukládat různé hodnoty, označujeme jako proměnné.

- ☞ Proměnné, třídy, zprávy a další entity, s nimiž v programu pracujeme, označujeme (pojmenováváme) prostřednictvím *identifikátorů*.
- ☞ V Javě mohou identifikátory obsahovat písmena (včetně písmen s diakritikou nebo např. japonských znaků), číslice a znaky „_“ (podtržítko) a „$“ (dolar). Nesmí začínat číslicí.
- ☞ Velbloudí notace je způsob zápisu, při němž se několikaslovný název píše dohromady bez mezer jako jedno slovo, přičemž každé slovo názvu začíná velkým písmenem a ostatní písmena jsou malá – např. `StrčPrstSkrzKrk`.
- ☞ Názvy tříd píšeme podle konvence velbloudí notací s prvním písmenem velkým – např. `StrčPrtstSkrzKrk` nebo `TřídaSDlouhýmNázvem`.
- ☞ Názvy metod a proměnných píšeme podle konvence velbloudí notací s prvním písmenem malým – např. `metodaSDlouhýmNázvem`.
- ☞ Java patří mezi jazyky, které v identifikátorech rozlišují velká a malá písmena.
- ☞ Zvláštní vlastnosti některých tříd můžeme v diagramu tříd vyznačit pomocí stereotypů, což jsou texty uzavřené ve «francouzských uvozovkách».

1.3 Testovací třída

- ☞ Vývojová prostředí bývají vybavena nástroji usnadňujícími tvorbu tzv. *jednotkových testů* (unit tests). V našich programech používáme k tomuto účelu knihovnu *JUnit*.
- ☞ Při využívání této knihovny se doporučuje definovat pro každou testovanou třídu její vlastní testovací třídu.
- ☞ Všechny testy v testovací třídě předpokládají, že při jejich spuštění je již připraven testovací přípravek.
- ☞ Testovací přípravek vytváří speciální metoda, která vytvoří potřebné objekty, uloží je do zásobníku odkazů a případně provede další potřebné inicializační akce.

1.4 Práce s daty

- ☞ Zasláním zprávy můžeme požadovat nejenom provedení nějaké akce, ale také např. vrácení hodnoty.
- ☞ Každá hodnota, kterou v programu použijeme, musí mít definován svůj *datový typ*.
- ☞ Datové typy dělíme v Javě na primitivní a objektové.
- ☞ Java definuje *primitivní typy* `boolean`, `char`, `double`, `int`, `long`, `byte`, `short` a `float`.
- ☞ *Objektové datové typy* jsou definovány jako třídy.
- ☞ Třídy, které jsou určeny k všeobecnému použití, jsou zahrnovány do tzv. API, což je zkratka z anglického *Application Program(ming) Interface*.
- ☞ Zprávy mohou mít *parametry*. Každý parametr má (stejně jako ostatní druhy dat) vždy definován svůj datový typ.
- ☞ Žádáme-li objekt zasláním zprávy o nějakou hodnotu, říkáme, že metoda realizující odpověď na danou zprávu, vrátí požadovanou hodnotu. Tato hodnota bývá označována jako *návratová hodnota* dané metody.

- Některé třídy žádáme o odkazy na jejich instance prostřednictvím tzv. *továrních metod*. Tovární metody mohou (na rozdíl od konstruktorů) vracet pokaždé odkaz na týž objekt.
- *Návrhové vzory* jsou doporučení, jak řešit některé typické, často se vyskytující úlohy. Jsou programátorskou obdobou matematických vzorečků, do nichž se místo čísel dosazují třídy, objekty a jejich metody.
  - Návrhový vzor *Knihovní třída* (*Utility class)* specifikuje, jak definovat třídu, která slouží pouze jako schránka na statické metody. Nepotřebuje instance, a proto by měla mít nepřístupný konstruktor.
  - Návrhový vzor *Statická tovární metoda* (*Static factory method*) doporučuje v případech, kdy třída chce mít lepší kontrolu nad počtem vytvořených instancí a způsobem jejich tvorby, vytvořit statickou metodu (metodu třídy), která bude pro okolní objekty sloužit jako náhrada konstruktoru.
  - Návrhový vzor *Jedináček* (*Singleton*) ukazuje, jak definovat třídu, která bude mít právě jednu instanci.
  - Návrhový vzor *Výčtový typ* ukazuje, jak definovat třídu, která bude mít předem známou a dále již neměnnou množinu instancí.

1.5 Výlet do nitra objektů

- Objekty si pamatují svůj stav prostřednictvím *atributů*. V anglické literatuře o jazyku Java bývají atributy často označovány jako pole (*field*).
- Objekty mají tři druhy členů:
  - *Datové členy* – atributy.
  - *Funkční členy* – metody.
  - *Typové členy* – datové typy definované uvnitř těchto typů.
- Vedle atributů svých instancí může třída definovat i atributy třídy. Atributy třídy uchovávají informace společné pro všechny instance dané třídy.
- O hodnoty uložené ve veřejných atributech můžeme žádat zadáním názvu dané oslovovaného objektu následovaného tečkou a názvem příslušného atributu. U instancí se za název objektu považuje název proměnné, v níž je uložen odkaz na objekt.
- Hodnoty lze získávat i zavoláním vhodné metody. Zavoláme-li tuto metodu na místě, kde předáváme hodnotu parametru, předá se jako hodnota parametru návratová hodnota volané metody.
- Potřebujeme-li pracovat se skupinou hodnot, bývá často výhodné definovat třídu, jejíž instance budou ve svých atributech uchovávat právě onu skupinu hodnot. Instance, jejichž jediným účelem je sloužit k uchování a přenášení skupiny hodnot, označujeme jako *přepravky*.
- V jazyku Java mohou metody vracet pouze jedinou hodnotu. Potřebujeme-li vrátit více hodnot současně, definujeme pro ně přepravku, do níž je uložíme a kterou daná metoda vrátí jako svoji (jedinou) návratovou hodnotu.

1.6 Programátorská dokumentace

- Dokumentace třídy se zobrazuje jako webová stránka, která začíná popisem třídy, za nímž následují tabulky se stručným popisem důležitých entit, a za těmito tabulkami pak následuje podrobný popis entit dané třídy.
- Identifikátory entit (datových typů, atributů, metod) slouží jako hypertextové odkazy, které nás přenesou k podrobné dokumentaci entity, na jejíž identifikátor jsme klepli.
- Textové podklady pro programátorskou dokumentaci jsou součástí zdrojového kódu.
- Součástí JDK je program nazvaný `javadoc`, který projde zadané zdrojové kódy, vyhledá v nich programátorskou dokumentaci a vytvoří z nich sadu webových stránek s jejich programátorskou dokumentací.
- V takto vytvořené dokumentaci můžeme prostřednictvím hypertextových odkazů přecházet i mezi jednotlivými datovými typy, a v případě správně nastavené konfigurace počítače se můžeme takto přenést i do dokumentace tříd standardní knihovny.
- Dokumentace standardní knihovny je sice k dispozici on-line, ale je vhodné ji mít staženou, abychom nebyli vázáni na přístup k internetu.

1.7 Rozhraní × `interface`

- Schopnost jazyka, umožnit objektům různých typů vydávat se za instanci nějakého společného rodičovského typu označujeme jako polymorfizmus.
- Z předchozí vlastnosti vyplývá schopnost objektů vydávat se v různých situacích za instance různých datových typů.
  - Okolní program přistupuje k danému objektu podle toho, za čí instanci se vydává.
  - Reakce objektu na zprávu nezávisí na tom, za čí instanci si objekt vydává, ale pouze na tom, čí instancí doopravdy je.
- Třídu, jejíž konstruktor vytvořil danou instanci, označujeme jako *mateřskou třídu* dané instance.
- *Deklarace* je část kódu, v níž předem oznamujeme některé vlastnosti vytvářeného programu.
- *Definice* je část kódu, v níž něco doopravdy vytváříme.
- Každá entita programu má dvě složky – *rozhraní* a *implementaci*.
  - Rozhraní dané entity specifikuje, co daná entita umí a jak s ní může okolní program komunikovat.
  - Implementace má na starosti, aby daná entita dělala přesně to, co její rozhraní slibuje.
- I rozhraní má dvě složky – *signaturu* a *kontrakt*.
  - Termínem signatura označujeme všechny vlastnosti, které může zkontrolovat překladač.
  - Termínem kontrakt označujeme vlastnosti dané entity, které překladač zkontrolovat nemůže (nemůžeme je deklarovat) a jejichž dodržení je zodpovědností programátora.
- *Interfejs* (anglicky `interface`) je druh datového typu, který pouze deklaruje své rozhraní a nemá žádnou implementaci.

- ☞ Interfejs nemůže mít vlastní instance. Za jeho instance se však mohou vydávat instance tříd, které daný interfejs *implementují*.
- ☞ Aby se instance třídy mohly vydávat za instance interfejsu, musí se třída k implementaci daného interfejsu veřejně (explicitně) přihlásit. Překladač pak zkontroluje, že třída opravdu implementuje vše, co implementovaný interfejs deklaruje.
- ☞ Hovoříme-li proto o instancích interfejsu, myslíme tím instance některé z tříd, které toto rozhraní implementují.
- ☞ V diagramu tříd se implementace interfejsu znázorňuje čárkovanou šipkou s trojúhelníkovou hlavičkou.
- ☞ V případech, kdy nehrozí nedorozumění, můžeme pro interfejs používat termín *rozhraní*.
- ☞ Najdeme-li u několika datových typů společné schopnosti, díky nimž bychom s nimi mohli pracovat jednotně, je vhodné definovat společný obecnější typ, který bude všechny tyto schopnosti deklarovat.
- ☞ Definujeme-li tento obecnější typ jako `interface`, který budou zmíněné datové typy implementovat, můžeme pak definovat metody, jejichž parametrem budou instance daného interfejsu, a které proto budou moci „obsluhovat“ instance libovolného z implementujících datových typů.
- ☞ Metody, jejichž parametry jsou instancemi nějakého rozhraní, jsou připraveny na spolupráci s instancemi libovolné třídy implementující dané rozhraní, a to i tehdy, pokud daná třída vznikne teprve někdy v budoucnu, takže o ní při definici oněch metod ještě nikdo neví.
- ☞ V této učebnici budeme používat konvenci, podle níž budeme identifikátory interfejsů doplňovat předponou (prefixem) `I` – např. `IPosuvný`.
- ☞ Návrhový vzor *Služebník (Servant)* ukazuje, jak přidat funkcionalitu skupině tříd, aniž bychom museli do každé z nich přidávat téměř shodnou metodu.
- ☞ Třída, jejíž instance vystupují jako služebníci, mívá přidružené rozhraní, které specifikuje, co musí umět instance, kterou má služebník „obsluhovat“.
- ☞ Java umožňuje definovat metody s proměnným počtem parametrů.

1.8 Pokročilá práce s rozhraním

- ☞ V některých případech se pro zadání potřebné hodnoty žádný ze standardních objektů nehodí. Návrhový vzor *Prázdný objekt* (*Null Object*) navrhuje řešit dané situace tak, že se definuje speciální, zdánlivě „nesmyslný“ objekt, který se použije pokaždé, když potřebná hodnota nedává smysl (např. barva květované sukně). Programátorům řešícím reakci na danou hodnotu pak odpadne řada přemýšlení a potenciálních chyb z přehlédnutí.
- ☞ Třída může implementovat několik interfejsů současně.
- ☞ Sady metod deklarované jednotlivými interfejsy se mohou překrývat.
- ☞ Při definici interfejsů s překrývajícími se sadami metod je třeba dbát na to, aby metody se stejnou signaturou měly i stejný kontrakt; jinak se prudce zvyšuje riziko budoucích problémů.
- ☞ Občas máme v projektu interfejs, který vyžaduje po implementujících třídách rozsáhlejší sadu metod, než kolik potřebuje definovaná metoda. V takovém případě můžeme definovat sadu minimálních „jednoúčelových“ interfejsů a nechat třídy implementovat

současně několik z nich. V metodě pak definujeme parametr jako instanci toho rozhraní, jež deklaruje právě to, co daná metoda potřebuje.

- ☞ Zvyšující se počet implementovaných interfejsů může poněkud znepřehledňovat program. Jednou z možností jeho snížení je zavedení *dědění rozhraní*.
- ☞ Potomek automaticky dědí rozhraní svého rodiče. Rozhraní potomka je pak sjednocením zděděného rozhraní případných dalších deklarací.
- ☞ Pro popis dědění se zavádí dvojice termínů *předek×potomek*, *rodičovský×dceřiný* typ, *základní×odvozený* typ, *nadtyp×podtyp*.
- ☞ Instance potomka se může kdykoliv vydávat za instanci svého předka.
- ☞ Implementace rozhraní je zvláštní druh dědění; v něm potomek (třída) implementuje metody deklarované v rozhraní svého předka – implementovaného rozhraní.
- ☞ Nutnost implementovat několik rozhraní současně lze zabezpečit definicí interfejsu, jenž bude společným potomkem všech interfejsů, které je třeba implementovat.
- ☞ Třídy, které se přihlásí k implementaci potomka, automaticky implementují všechny jeho předky.
- ☞ Interfejs nemusí deklarovat žádné metody. V řadě případů je užitečné, když interfejs pouze deklaruje nějaký kontrakt. Rozhraní (`interface`), které nedeklaruje žádnou metodu, nazýváme *značkovací*. Implementací tohoto rozhraní třída pouze slibuje, že její instance budou dodržovat jeho kontrakt.
- ☞ Návrhový vzor *Prototyp* radí, aby objekt poskytl specializovanou tovární metodu, která na požádání dodá jeho kopii. Je však na rozhodnutí dané metody, co bude za kopii objektu vydávat a jak tento objekt získá.
- ☞ *Prototyp* využijeme s výhodou v situacích, kdy je konstruktor nedostupný či obtížně použitelný.
- ☞ Pro výše zmíněnou kopii je důležité, aby se chovala stejně jako její vzor. Její skutečný mateřský typ se však může lišit.
- ☞ Konstrukce `interface` a pravidla implementace rozhraní nám dovoluji elegantně simulovat situace, kdy se objekt vydává za někoho jiného, než ve skutečnosti je.
- ☞ Objekt oslovující jiný objekt je na osloveném objektu závislý. Jakmile se změní definice oslovovaného objektu, musíme zkontrolovat, jestli tato změna nevyžaduje také změnu v definici oslovujícího objektu. Aplikace tohoto pravidla může vyvolat dominový efekt vedoucí k úpravám velké části projektu.
- ☞ Při vzájemné komunikaci mnoha objektů roste počet vzájemných závislostí a tím i pravděpodobnost vzniku dominového efektu při nutnosti provést nějakou změnu.
- ☞ Návrhový vzor *Prostředník* (anglicky *Mediator*) ukazuje, jak lze zavedením prostředníka výrazně snížit počet závislostí.
- ☞ Pouhá definice prostředníka sice sníží počet závislostí, ale nezamezí dominovému efektu.
- ☞ Princip *injekce závislostí* (anglicky *dependency injection*) doporučuje, aby objekt, který musí oslovovat řadu jiných objektů, deklaroval požadované rozhraní a komunikoval pouze s instancemi tohoto rozhraní. Ostatní objekty se tak musejí naopak přizpůsobit jemu a jeho rozhraní (vnutí [injektuje] jim tak závislost na sobě).
- ☞ Návrhový vzor *Pozorovatel* (anglicky *Observer*) řeší situaci, kdy objekt čeká na výskyt nějaké události, aby na ni mohl zareagovat. Podle tohoto vzoru by se objekt čekající na

zprávu (pozorovatel) měl zaregistrovat u objektu, který mu při výskytu dané události pošle zprávu, že událost nastala.

- Pro návrhový vzor *Pozorovatel* se používají i názvy *Posluchač* (anglicky *Listener*) a *Vydavatel-Předplatitel* (anglicky *Publisher-Subscriber*).
- Návrhový vzor *Pozorovatel* realizuje tzv. *Hollywoodský princip* charakterizovaný sloganem: „Nevolejte nám, zavoláme vám.“

Doprovodný projekt

- Při přemisťování zobrazovaných tvarů v našich počátečních projektech se tyto tvary vzájemně odmazávaly. Aby se „díry zacelily“, musely by přemisťované objekty ostatním sdělit, že se přemisťují, aby se tyto mohly překreslit.
- Nový projekt nahradil pasivní plátno aktivním správcem plátna, který zabezpečuje, že každý zobrazovaný objekt dostane ve správný okamžik šanci se nakreslit. Instance třídy `CanvasManager` představuje prostředníka, který zabezpečuje komunikaci mezi zobrazovanými objekty.
- Instance třídy `CanvasManager` (správce plátna) implementuje návrhový vzor *Pozorovatel*: všechny objekty, které chtějí být v novém projektu zobrazovány na plátně, musejí implementovat rozhraní `IPaintable` a jako instance tohoto rozhraní se musejí u správce plátna registrovat.
- Pokaždé, když zobrazovaný objekt změní svůj vzhled, musí požádat správce plátna, aby nechal plátno překreslit.
- Při překreslování plátna posílá správce postupně všem registrovaným objektům zprávu (volá metodu) `paint(Painter painter)`, v jejímž parametru jim předá kreslítko (malíře), jež je jediným objektem, který ví o plátně a umí na ně něco nakreslit.
- Při převodu testovacích tříd ze starého projektu s obyčejným plátnem do projektu se správcem plátna je třeba upravit definici testovacího přípravku a zařídit, aby se zobrazované objekty přihlásily u správce plátna.

1.9 Dědění tříd [RUP1]

- Existují tři druhy dědění:
  - Přirozené (nativní) dědění, které odráží naši představu o tom, které objekty jsou speciálním případem jejich obecnějších druhů.
  - Dědění typu, které odpovídá dříve probranému dědění rozhraní.
  - Dědění implementace, při němž potomek dědí i definice metod specifikujících reakce na obdržené zprávy.
- Dobře navržený program má všechny tyto tři aspekty dědění v souladu.
- Instance potomka musí být schopna kdykoliv plnohodnotně vystupovat v roli instance předka. Toto pravidlo bývá označováno jako *Substituční princip Barbary Liskové* (*Liskov Substitution Principle* – LSP).
- Rodičovská třída umožňuje svým potomkům vytknout společný kód do společného rodiče.

- Při uvažování o tom, že bychom pro skupinu tříd definovali společného rodiče, musíme nejprve zvážit, jsou-li opravdu všechny třídy z této skupiny speciálním případem nějakého obecnějšího objektu, jehož instance bude vytvářet plánovaná rodičovská třída. Jinými slovy: nesmíme v zájmu zdědění implementace porušit LSP.
- Definici společných rodičovských tříd musíme někdy rozdělit do několika vrstev.
- Třída, která dědí od jiné třídy, se chová, jakoby vybavila své instance atributem s odkazem na instanci rodiče označovanou jako rodičovský podobjekt.
- Kdykoliv objektu někdo pošle zprávu, zavolá metodu, která má na tuto zprávu reagovat. Pokud oslovený objekt takovou metodu nemá, přepošle tuto zprávu svému rodičovskému podobjektu, který učiní totéž.
- Dceřiná třída se může rozhodnout, že se jí rodičovská reakce na nějakou zprávu nelíbí a že si proto definuje vlastní metodu, která překryje metodu zděděnou. Kdykoliv pak její instanci pošle někdo danou zprávu, bude vždy vyvolána nově definovaná metoda.
- Překryté metody jsou z okolí objektu nedostupné. Volat je může pouze objekt sám, a to tak, že přímo osloví svůj rodičovský podobjekt.
- Rodičovský podobjekt je pro okolní svět nedostupný, a proto nemůže nikdy dostat zprávu od okolního programu.
- Posílá-li rodičovský podobjekt překrytelnou zprávu sám sobě (tj. volá-li svoji překrytelnou metodu), posílá ji ve skutečnosti svému majiteli – dceřinému objektu. Pokud tento objekt rodičovskou metodu překryl, zavolá logicky svoji „lepší“ verzi.
- Jakmile třída definuje vlastní verzi metody (překryje rodičovskou metodu), její instance už vždy použijí metodu definovanou jejich mateřskou třídou, a to i tehdy, budou-li se vydávat za instance svého předka.
- Všechny třídy mají v Javě právě jednoho předka. Jedinou výjimkou je třída `Object`, která je společným rodičem všech tříd a která žádného předka nemá.
- Třída, která se explicitně nepřihlásí k žádnému svému bezprostřednímu rodiči, je automaticky potomkem třídy `Object`.
- Očekává-li nějaká zpráva, resp. metoda parametr typu `Object`, můžeme ji jako parametr předat cokoliv včetně hodnoty primitivních typů, kterou za nás před předáním zabalí překladač do vhodného objektu.
- Vrací-li nějaká zpráva, resp. metoda hodnotu typu `Object`, víme o ní pouze to, že není primitivního typu.
- Abstraktní třída stojí svými vlastnostmi někde mezi klasickou třídou a interfejsem:
  - Nemusí mít implementované všechny metody, ale pak kvůli tomu nemůže vytvářet svoje instance jinak, než jako rodičovské podobjekty svých potomků.
  - Může mít definované atributy a metody, ale pak kvůli tomu smí mít pouze jediného třídního předka. Může ale implementovat libovolný počet rozhraní.
- Třídy, které mohou vytvářet klasické objekty, označujeme jako konkrétní.
- Metodu, která je pouze deklarovaná a není definovaná, označujeme jako abstraktní. Definovanou metodu označujeme jako konkrétní.
- V interfejsech jsou všechny metody automaticky abstraktní.
- Abstraktní třídu využijeme např. tehdy, vyžaduje-li implementované rozhraní definici metody, o níž víme, že ji bude každý potomek definovat po svém.

- ☞ Abstraktní třída může deklarovat i své vlastní abstraktní metody, jejichž definici ponechá na potomcích.
- ☞ Návrhový vzor *Šablonová metoda* radí definovat společnou funkcionalitu metody v předkovi s tím, že části algoritmu, v nichž se názor na jejich provedení u jednotlivých potomků liší, se definují jako překrytelné (často abstraktní) metody, pomocí nichž mohou potomci vyjádřit svůj vlastní názor na to, jak se má daná část algoritmu realizovat.

1.10 Vytváříme vlastní třídu

- ☞ Přeložené soubory mají stejný název jako definovaná třída a mají příponu class.
- ☞ Každá třída má svůj vlastní class-soubor.
- ☞ Definice třídy sestává z hlavičky a těla.
  - ☞ Hlavička obsahuje modifikátory blíže specifikující vlastnosti třídy, klíčové slovo `class` a název třídy.
  - ☞ Tělo následuje za hlavičkou, je uzavřeno ve složených závorkách a obsahuje definice všech členů dané třídy.
- ☞ Zdrojový kód veřejné třídy je uložen v souboru, který má stejný název jako třída (musí se dodržet i velikost písmen) a má příponu java.
- ☞ Pro mezeru a znaky tabulátoru, konce řádku a konce stánky používáme souhrnný název bílé znaky.
- ☞ V místě, kde smí být v programu mezera, může být i libovolný počet bílých znaků.
- ☞ Mezeru můžeme v programu vložit kamkoliv s výjimkou vnitřku identifikátorů a literálů.
- ☞ Java nevynucuje žádné pořadí deklarací a definic jednotlivých členů.
- ☞ Po každé úpravě zdrojového kódu je třeba třídu před jejím prvním použitím přeložit.
- ☞ Proces, při němž hledáme a odstraňujeme chyby v programu, se nazývá *ladění*.
- ☞ V programech rozeznáváme tři druhy chyb: *syntaktické*, *běhové* a *logické* (*sémantické*).
  - ☞ Syntaktické chyby objeví překladač při překladu,
  - ☞ běhové způsobí havárii programu za běhu,
  - ☞ logické se projeví nesprávným výsledkem, který však nemusíme na první pohled poznat.
- ☞ Konstruktor je zvláštním druhem metody. Má za úkol inicializovat právě vytvořenou instanci.
- ☞ Definice konstruktoru sestává z hlavičky a těla. Hlavička obsahuje modifikátory následované typem inicializované instance, který je shodný s názvem třídy, a seznamem parametrů uzavřeným v kulatých závorkách. Nemá-li konstruktor žádné parametry, budou závorky prázdné.
- ☞ Každá třída má konstruktor. Pokud programátor žádný nedefinuje, doplní za něj překladač ten nejjednodušší možný.

1.11 Přidáváme parametry

- Pro efektivní vývoj programů je výhodné napsat nejprve testy a teprve pak testovaný program. Celý vývoj pak má jediný cíl: rozchodit testy.
- Výše popsanou metodiku nazýváme *Vývoj programů řízený testy*, anglicky *Test Driven Development* – TDD.
- Při testech využíváme *testovací přípravek* (test fixture), což jsou definované počáteční podmínky každého testu.
- Parametr deklarujeme tak, že uvedeme jeho typ následovaný názvem (identifikátorem) parametru, prostřednictvím nějž se k němu budeme v těle metody obracet. U objektových parametrů uvádíme jako jejich typ název typu (třídy či rozhraní), za jehož instanci se vydávají.
- Je-li pro nás výhodné zavolat v konstruktoru jiný konstruktor téže třídy, zavoláme jej tak, že napíšeme klíčové slovo `this` následované seznamem parametrů v kulatých závorkách. Překladač odvodí, který konstruktor chceme zavolat, podle počtu a typu parametrů.
- Volání konstruktoru prostřednictvím `this` musí být úplně prvním příkazem těla konstruktoru, před ním smějí být již pouze mezery a komentáře.
- Výskyt více stejně pojmenovaných metod lišících se pouze počtem a/nebo typem jednotlivých parametrů označujeme jako přetěžování (anglicky overloading).
- Zaslání zprávy objektu realizujeme zavoláním odpovídající metody.
- Metody se musí navzájem lišit svým jménem (identifikátorem) a/nebo seznamem typů parametrů. Metody (včetně konstruktorů), které se liší pouze počtem a/nebo typem parametrů označujeme za *přetížené*.

1.12 Přidáváme atributy a metody

- Atributy deklarujeme v těle třídy, avšak mimo těla jejich metod.
- Není vhodné prokládat v kódu definice atributů a metod. Atributy by měly být definovány pohromadě a metody také.
- Většina programátorů se drží konvence, podle níž se atributy umisťují na začátek definice třídy a metody až za ně.
- Deklarace atributu sestává z:
  - nepovinných modifikátorů blíže specifikujících některé jeho vlastnosti,
  - typu atributu,
  - názvu atributu,
  - nepovinného přiřazení počáteční hodnoty.
- Dokud atributu nepřiřadíme nějakou hodnotu sami, je jeho hodnotou nula, resp. `false`, resp. `null`.
- Deklarace několika atributů stejného typu můžeme sloučit do jedné deklarace. Taková deklarace pak začíná volitelnými modifikátory následovanými názvem typu. Za něj vložíme seznam identifikátorů deklarovaných atributů oddělených čárkami. Součástí deklarací mohou být i případné inicializace.
- Nemáme-li opravdu pádný důvod zavést atribut jinak, měl by být soukromý, tj. měl by mít označen jako `private`.

- Soukromé by měly být i metody určené pouze pro vnitřní potřebu metod dané třídy.
- Atributy, jejichž hodnoty se nebudou v průběhu života objektu měnit, by měly být definovány jako konstanty, tj. označeny modifikátorem `final`, aby mohl překladač kontrolovat, že jejich hodnotu omylem neměníme.
- Přiřadit hodnotu konstantě nám překladač dovolí jenom jednou – buď přímo v deklaraci, nebo v konstruktoru. Jakmile ji inicializujeme, už nám překladač nedovolí její hodnotu změnit.
- Definice metody sestává z hlavičky a těla. *Hlavička* obsahuje modifikátory následované typem návratové hodnoty, názvem metody a seznamem parametrů uzavřeným v kulatých závorkách.
- V hlavičce metody se musí vždy uvádět typ návratové hodnoty. Když metoda nic nevrací, uvádí se `void`.
- Za názvem metody musí být vždy seznam parametrů v závorkách. Nemá-li metoda žádné parametry, uvedou se alespoň prázdné závorky.
- Jednotlivé verze přetížených metod se liší počtem a/nebo typy parametrů v závorkách za názvem metody.
- Pro přesný zápis pravidel, podle nichž se vytváří (zapisuje) vysvětlovaná konstrukce, používáme *syntaktické definice* a/nebo *syntaktické diagramy*.
- Metody vracejí požadované hodnoty tak, že jako poslední vykonávaný příkaz uvedou příkaz `return` následovaný výrazem, jehož hodnotu metoda vrací.
- U objektů rozlišujeme atributy a vlastnosti.
  - Atribut je interní záležitost objektu zavedená proto, aby objekt mohl plnit požadovanou funkci.
  - Vlastnost je informace o stavu objektu, kterou můžeme zjistit nebo nastavit.
- Vlastnosti zjišťujeme a nastavujeme prostřednictvím přístupových metod.
- Každá zpráva musí mít svého adresáta. Část kódu specifikující adresáta zprávy označujeme jako kvalifikaci.
- Klíčové slovo `this` zastupuje odkaz na instanci, v jejíž metodě se nacházíme. Kdykoliv potřebuje metoda použít svoji vlastní instanci, použije odkaz ve skrytém parametru `this`.
- Chceme-li zdůraznit, že se obracíme na atribut či metodu té instance, jejíž metodu právě definujeme, kvalifikujeme ji klíčovým slovem `this`.
- Rozlišujeme explicitní kvalifikaci zadanou programátorem a implicitní kvalifikaci, kterou si domyslí překladač. Nezadá-li programátor explicitní kvalifikací, pokusí se překladač doplnit `this`, a pokud takto upravený program vyhovuje syntaktickým pravidlům, považuje původní program za správný.
- Měla-li by implicitní kvalifikace kolidovat s požadovanou, musí programátor zadat kvalifikaci explicitně.
- Explicitně zadaná kvalifikace má vždy přednost.
- Objekty můžeme kvalifikovat zadáním názvu proměnné obsahující odkaz na daný objekt nebo posláním, zprávy, která vrací požadovaný objekt.

1.13 Pokročilejší práce s daty

- V případě potřeby můžeme uvnitř metody deklarovat lokální proměnné. Proměnné se deklarují obdobně jako atributy s tím rozdílem, že jediným povoleným modifikátorem je `final`.
- Lokální proměnnou nelze použít, dokud se jí nepřiřadí nějaká hodnota.
- Parametr vystupuje jako lokální proměnná, kterou inicializuje volající kód.
- Má-li metoda parametr či lokální proměnnou se stejným názvem jako atribut, musíme v metodě atribut kvalifikovat. Identifikátor bez kvalifikace totiž v takovéto metodě označuje onen stejně pojmenovaný parametr či lokální proměnnou.
- Po ukončení metody jsou všechny její lokální proměnné ztraceny. Potřebuje-li si metoda něco pamatovat mezi svými spuštěními, musí si to uložit do nějakého atributu.
- Atributy a metody třídy definujeme tak, že mezi jejich modifikátory uvedeme klíčové slovo `static`. Atributy a metody třídy bývají proto často označovány jako *statické*.
- Statické metody nejsou spojeny s žádnou instancí. Proto v nich není možné použít klíčové slovo `this`, protože by nevěděly, na co odkazuje.
- Má-li být hodnota atributu, lokální proměnné či parametru konstantní, uvedeme mezi modifikátory klíčové slovo `final`.
- Statickým konstantám je třeba přiřadit jejich hodnotu již v deklaraci, nestatickým konstantám je možno přiřadit počáteční hodnotu v deklaraci nebo v těle konstruktoru.
- Metody mohou mít i parametry objektových typů. Při zadávání hodnot takovýchto parametrů v prostředí *BlueJ* můžeme využít možnosti zadat název proměnné klepnutím na příslušný odkaz v zásobníku odkazů.
- Jako *literály* označujeme hodnoty přímo zapsané do programu, tj. přímo zadaná čísla, znaky, textové řetězce a konstanty `true`, `false` a `null`.
- V celých číslech můžeme pro zpřehlednění používat znak podtržení – např. 123_456_789. Znak podtržení nesmí být ani prvním ani posledním znakem čísla.
- Celá čísla začínající nulou chápe překladač z historických důvodů jako čísla zapsaná v osmičkové soustavě.
- Literály typu `long` mají za číslem příponu `L` nebo `l`. Použití malého `l` se nedoporučuje, aby se nepletlo s jedničkou.
- Datové typy `byte` a `short` nemají své vlastní literály a používají se pro ně přetypovaná čísla typu `int`.
- V desetinných číslech se místo desetinné čárky používá desetinná tečka.
- Reálné číslo můžeme zapsat v normálním nebo semilogaritmickém tvaru.
- Semilogaritmický tvar bývá někdy označován jako vědecký.
- Číslo zapsané v semilogaritmickém tvaru sestává z mantisy (celé nebo desetinné číslo) následované znakem `E` nebo `e` a celočíselným exponentem uvádějícím kolikátou mocninou deseti se má mantisa vynásobit.
- Chceme-li, aby se literál celého čísla uložil jako číslo typu `double`, zapíšeme za ně příponu `D` nebo `d`. Tento znak můžeme zapsat i za reálná čísla, i když je tam zbytečný, protože ta jsou implicitně typu `double`.
- Chceme-li zadat číslo typu `float`, přidáme za ně příponu `F` nebo `f`.

- Znakové literály zapisujeme mezi apostrofy.
- Pro nejčastěji používané obtížně zadatelné znaky se používají předdefinované escape sekvence tvořené zpětným lomítkem a dalším znakem. Java zavádí sekvence `\b \t \n \f \r \" \' \\`
- Pro znaky zadávané svým kódem se používá escape sekvence `\uHHHH`, kde `HHHH` je čtyřmístné hexadecimální číslo představující kód daného znaku.
- Dlouhý řetězec můžeme složit z několika částí, které „sečteme“ do výsledného řetězce.
- Řetězec musí být v programu celý na jednom řádku. Je-li příliš dlouhý, můžeme jej rozdělit na několik kratších, jednořádkových řetězců, které „sečteme“.
- Anotací `@Override` oznamujeme překladači, že definujeme vlastní verzi zděděné metody. Tato verze má překrýt zděděnou metodu, a překladač by proto měl zkontrolovat, jestli jsme neudělali chybu a jestli rodič opravdu má překrytelnou metodu s danou signaturou.

1.14 Komentáře a dokumentace

- U tříd rozeznáváme:
  - rozhraní, tj. to, co o sobě třída zveřejní a na co se mohou její uživatelé spolehnout,
  - implementaci, tj. to, jak třída zařídí, že umí to, co vyhlásila v rozhraní.
- Implementační detaily bychom měli před okolím skrývat, aby nebylo možno funkčnost třídy a jejích instancí ohrozit.
- Vedle explicitně deklarované části rozhraní, tj. hlaviček veřejných metod, je součástí rozhraní i tzv. *kontrakt* popisující rysy, které není možno přímo specifikovat prostředky jazyka, a je třeba je uvést v doprovodné dokumentaci.
- Jazyk Java používá dva druhy komentářů: obecný, který je ohraničen komentářovými závorkami `/*` a `*/`, a řádkový, který začíná znaky `//` a končí spolu s koncem řádku.
- Komentář můžeme napsat kdekoliv, kde můžeme napsat mezeru.
- Obecný komentář začínající znaky `/**` je chápán jako dokumentační. Zapisují se do něj informace užitečné pro budoucí uživatele dané třídy, metody, konstanty atd.
- Vývojové prostředí Javy obsahuje program `javadoc`, který projde zdrojový kód a z dokumentačních komentářů vytvoří standardní dokumentaci.
- Dokumentační komentáře mohou vedle prostého textu obsahovat i HTML značky (tag).
- Java zavádí několik speciálních značek začínajících znakem „@“, které slouží k lepšímu popisu některých rysů dokumentovaných konstrukcí, např. parametrů či návratových hodnot dokumentovaných metod.
- Dokumentační komentáře musíme napsat těsně před dokumentovanou entitu (třídu, atribut, metodu).

1.15 Operace a operátory

- Prázdný řetězec je standardní objekt – řetězec, který neobsahuje žádné znaky. Naproti tomu prázdný odkaz je informace o tom, že daná proměnná na žádný objekt neukazuje.
- Operace je to, co se provede.

- Operátor je znak nebo skupina znaků, které oznamují, jaká operace se má v daném místě provést.
- Operand je výraz, s nímž se provádí operace.
- Arita operace říká, kolik v ní vystupuje operandů. Java používá operace nulární, unární, binární a ternární.
- Operace násobení má ve výrazech přednost před sčítáním.
- Potřebujeme-li při skládání („sčítání“) textového řetězce provést nejprve nějakou operaci, jejíž výsledek se má stát součástí řetězce, je vhodné tuto operaci uzavřít do závorek.
- Typ výsledku binární aritmetické operace je určen typem obecnějšího z operandů.
  - Je-li jeden operand typu `String`, je výsledek typu `String`.
  - Jsou-li oba operandy číselné a jeden je reálné číslo, je výsledek reálný.
  - Jsou-li oba operandy celočíselné, ale jeden je „dlouhý“ (`long`), je výsledek „dlouhý“.
  - Jsou-li oba celočíselné typu `int` nebo menšího, je výsledek typu `int`.
- Jsou-li v podílu oba operandy celá čísla, je výsledek podílu celé číslo vzniklé oříznutím desetinné části reálného podílu.
- Operátor `%` (dělení modulo, zjišťování zbytku po dělení) je definovaný i pro reálná čísla. Znaménko výsledku je stejné jako znaménko čitatele.
- Operátory + a – mohou vystupovat i jako unární – pak ovlivňují znaménko hodnoty svého operandu.
- Přiřazovací operátor je binární operátor, který vrací jako hodnotu provedené operace přiřazovanou hodnotu.
- Složené přiřazovací operátory slučují binární operaci svých dvou operandů s přiřazením výsledku operace levému operandu.
- Operátor přetypování vrací hodnotu svého operandu jako hodnotu typu, na nějž jej přetypovávají.
- Přetypování rozeznáváme implicitní, které za nás dělá překladač, a explicitní, o které musíme požádat sami.
- Implicitně se přetypovává menší numerický typ na větší a potomek na předka. O ostatní druhy přetypování musíme v programu explicitně požádat.
- Každý primitivní typ má přiřazen svůj obalový objektový typ, který slouží k předání hodnoty primitivního typu metodám vyžadujícím parametr objektového typu.
- Obalové typy definují řadu užitečných metod. K nejpoužívanějším patří metody nazvané `parseXxx`, kde `Xxx` je název daného primitivního typu s prvním písmenem velkým. Tyto metody převádějí řetězec na hodnotu příslušného primitivního datového typu.
- Inkrementační a dekrementační operátory mohou být zapsány před svým operandem, anebo za ním. Podle umístění operátoru se liší provedená akce:
  - Je-li operátor umístěn ve výrazu před operandem, nejprve se provede operace, a pak se ve výrazu použije inkrementovaná, resp. dekrementovaná hodnota.
  - Je-li operátor umístěn ve výrazu za operandem, použije se ve výrazu aktuální hodnota operandu, který se následně inkrementuje, resp. dekrementuje.
- In/de-krementační operátory uvedené před operandem se označují jako prefixové, operátory uvedené za operandem jako postfixové. Alternativně se prefixové operátory označují jako pre(in/de)krementační, postfixové jako resp. post(in/de)krementační

- ☞ Rozhodneme-li se počítat vytvořené instance dané třídy, definujeme statický atribut, v němž si budeme pamatovat počet doposud vytvořených instancí, a konstantní instanční atribut, do nějž budeme ukládat aktuální stav statického atributu jako rodné (identifikační) číslo instance.
- ☞ Při aktualizaci stavu statického atributu z předchozího bodu můžeme využít preinkrementační operátor.
- ☞ Předávání hodnot mezi metodami prostřednictvím atributů není považováno za optimální způsob. Za výhodnější se považuje předání hodnot prostřednictvím parametrů.
- ☞ Vede-li porušení některé zásady k zpřehlednění programu, bývá vhodné ji porušit.
- ☞ Standardní výstup je objekt uložený v proměnné `System.out`, standardní chybový výstup je objekt uložený v proměnné `System.err`.
- ☞ Na standardní výstupy se tiskne zavoláním jejich metod `print(???)` nebo `println(???)`, přičemž druhá z nich na konci výstupu navíc odřádkuje.

Doprovodný projekt

- ☞ Pro jednoduché načítání a zveřejňování informací prostřednictvím dialogových oken lze v našem projektu s výhodou použít statické metody třídy `IO`.
- ☞ Třída `IO` obsahuje statické metody `inform`, `enter` a `confirm`, které umožňují jednoduché zadávání výstupních zpráv a požadavků na vstupní hodnoty prostřednictvím dialogových oken.

1.16 Definice testovací třídy

- ☞ Testovací třídy používané v našich projektech využívají služeb knihovny *JUnit*.
- ☞ Knihovna *JUnit* je de-facto standardem, který používá většina vývojových prostředí, programovacích jazyků a platforem.
- ☞ V testovací třídě chápe systém metody označené anotací `@Test` jako testy, které je schopen na požádání spustit.
- ☞ Podle konvence by měl název testovací metody začínat předponou `test` následované stručnou charakteristikou testu.
- ☞ Před každým testem se spustí metoda definovaná ve stejné třídě a označená anotací `@Before`. Tato metoda má za úkol vytvořit testovacího přípravek (test fixture), což je výchozí sada objektů, se kterou testy pracují. Podle konvence by se měla jmenovat `setUp`.
- ☞ Po každém testu se spustí metoda definovaná ve stejné třídě a označená anotací `@After`. Tato metoda má za úkol úklid po testu a podle konvence by se měla jmenovat `tearDown`.
- ☞ Testovací metody spolu s metodami pro vytvoření přípravku a úklid po testu musí být veřejné, bezparametrické a nesmí nic vracet (musí vracet `void`).
- ☞ Pro porovnání očekávané a obdržené hodnoty či jinak specifikovanou kontrolu stavu programu slouží sada metod `assertXxx(???)`, které prověří požadovaný stav a v případě nesouhlasu označí test jako neúspěšný.
- ☞ Testovací metody jsou standardní metody, které se mohou vzájemně volat.
- ☞ Při volání testovací metody z jiné metody se přípravek nevytváří.

1.17 Ladění programů

- Při hledáních chyby programu používáme:
  - Kontrolní tisky oznamující kudy program prošel a v jakém byl právě stavu.
  - Ladící program nazývaný většinou *debugger*, který nám umožní procházet programem krok za krokem a průběžně sledovat hodnoty jednotlivých proměnných.
- Při práci s ladícím programem nastavujeme v programu zarážky, kde má ladící program zastavit chod programu a ukázat nám jeho stav.
- Zarážku vložíme v editoru klepnutím do sloupce s čísly řádků na řádek, před jehož provedením se má program zastavit.
- Zarážku můžeme vložit pouze v případě, je-li třída přeložena.
- Zarážku musíme vložit na řádek, kterému je v class-souboru přiřazen nějaký kód.
- Při spuštění programu s nastavenými zarážkami běží program až k nastavené zarážce. Před ní se zastaví, a:
  - zobrazí zdrojový kód s vyznačeným řádkem, na jehož zarážce se zastavil,
  - zobrazí okno debuggeru, jež umožní bližší analýzu aktuálního stavu programu.
- Zarážky je možné nastavovat i rušit za běhu programu.
- Lokální proměnné se ve svém panelu zobrazí až v okamžiku, kdy je jim přiřazena nějaká hodnota.

1.18 Implementace rozhraní

- Definice interfejsu se od definice třídy liší ve dvou věcech:
  - V hlavičce je místo klíčového slova `class` použito klíčové slovo `interface`.
  - V těle interfejsu jsou zapsány pouze hlavičky metod ukončené středníkem.
- Stejně jako ve třídě se i v interfejsu zapisují před hlavičky metod dokumentační komentáře.
- Všechny metody interfejsu jsou automaticky veřejné, a proto se v jejich hlavičce nemusí uvádět modifikátor `public`.
- Všechny metody interfejsu jsou automaticky abstraktní, a proto se se v jejich hlavičce nemusí uvádět modifikátor `abstract`.
- V interfejsu nemůžete deklarovat metody s atributem `static`.
- V interfejsu lze definovat statické konstanty. V současné době se ale využití této možnosti nedoporučuje.
- V definici interfejsu je vhodné uvádět před hlavičkami zakomentovanou anotaci `@Override`.
- V definici interfejsu je vhodné v dokumentačních komentářích důsledně uvádět kompletní kontrakt.
- Při implementaci interfejsu je vhodné si dobře přečíst dokumentační komentáře metod, abychom nezapomněli dodržet vše, co kontrakt vyžaduje.
- Ve zdrojovém kódu se implementace rozhraní označuje v hlavičce třídy klauzulí `implements Xyz`, kde `Xyz` je implementované rozhraní, uváděnou za názvem třídy, např.

```
public class Světlo implements IPaintable
```

Tuto klauzuli budeme v našem kurzu označovat jako implementační dovětek.

- ☞ Při přizpůsobování importované třídy novému projektu postupujeme ve třech krocích:
  1. Překlad třídy bez výrazné snahy o splnění funkčnosti.
  2. Překlad testovací třídy.
  3. Zprovoznění testů.
- ☞ Oblíbená chybová hláška

  ... is not abstract and does not override abstract method ... in ...

  bývá způsobena tím, že třída neimplementuje všechny abstraktní metody svých rodičů, většinou metody implementovaných rozhraní.

Doprovodný projekt

- ☞ Při používání správce plátna nesmíme zapomínat přihlásit do správy všechny objekty, které se mají na plátně zobrazovat.
- ☞ Voláním metody `setSize(int,int)` nastavuje správce plátna zadanou políčkovou velikost plátna při aktuální velikosti políčka. Pro rafinovanější nastavení velikosti políček a plátna slouží metoda `setStepAndSize(init,int,int)`.
- ☞ Při používání správce plátna zůstávají obrazce na plátně až do doby, dokud někdo správce nepožádá o jejich odstranění.
- ☞ Při práci s komplexními objekty musíme dbát na to, aby se u správce přihlásil vždy celý objekt a ne jeho části. Stejně tak při odhlášení.

1.19 Samostatná aplikace – UFO

- ☞ Má-li být aplikace samostatně spustitelná, musí obsahovat třídu s veřejnou statickou metodou `main` s jediným parametrem typu `String[]`. Metoda nesmí nic vracet, tj. musí mít typ návratové hodnoty `void`.
- ☞ JAR-soubor je běžný ZIP soubor se složkou META-INF obsahující soubor MANIFEST.MF.
- ☞ Označíme-li při tvorbě JAR-souboru spustitelnou třídu, můžeme pak soubor používat jako spustitelnou aplikaci.

1.20 Refaktorace

- ☞ Návrhový vzor *Jedináček* (*Singleton*) ukazuje jak definovat třídu, která povoluje vytvoření pouze jediné instance.
- ☞ Odkaz na instanci jedináčka je uložen v atributu třídy, který je možno inicializovat hned v deklaraci atributu zavoláním příslušného konstruktoru.
- ☞ Třída s jedináčkem nesmí definovat veřejný konstruktor. Konstruktor musí být soukromý a k získání odkazu na jedináčka musí poskytovat tovární metodu.
- ☞ Tovární metody tříd s jedináčky se nejčastěji jmenují `getInstance`.
- ☞ Termínem *pachy v kódu* označujeme vlastnosti, které ztěžují další udržování kódu, především pak jeho budoucí modifikaci.

- Refaktorování je postup, při němž v drobných krocích upravujeme program tak, aby se zlepšila jeho architektura a tím se usnadnila jeho případná další vylepšení.
- Základem refaktorování je zásada, že úpravy musejí probíhat v opravdu malých krocích. Po každém kroku se spustí testy ověřující, že jsme tímto krokem nenarušili funkčnost programu.
- Mezi základní refaktorovací operace patří přejmenování entity programu (třídy, proměnné, metody, …) a vyjmutí části metody do samostatné metody.
- Metoda, která má více jak 8 příkazů, je podezřelá, že dělá několik věcí. Je vhodné se proto zamyslet nad tím, jestli by ji nebylo možno rozložit na několik metod jednodušších.
- V programovacích jazycích se používá předávání parametrů hodnotou a odkazem.
  - Při předávání hodnotou se předává kopie hodnoty parametru, a volající metoda netuší, co se s parametrem děje.
  - Při předávání parametrů odkazem se ve skutečnosti předává odkaz na místo v paměti, takže po skončení metody zde můžeme najít jinou hodnotu, než tam byla při jejím volání.
- Předávání parametrů odkazem narušuje zapouzdření.
- Java předává parametry metodám zásadně hodnotou.
- U objektových typů je za hodnotu parametru považována odkaz na předávaný objekt. U objektových parametrů se proto může stát, že po skončení metody má objekt jiné vlastnosti, než jaké měl jako předávaný parametr. Nemůže se však stát, že by metoda vrátila v parametru jiný objekt.
- Potřebujeme-li ve třídě používat parametry, jejichž hodnotu oslovená metoda změní, můžeme je zabalit do objektu jako jeho proměnné atributy. Metodě pak předáme tento obalový objekt a po skončení metody si v něm vyzvedneme změněné hodnoty.
- Potřebujeme-li v programu třídu, která bude používána pouze metodami jedné třídy, můžeme potřebnou třídu definovat uvnitř třídy, jejíž metody ji budou používat, jako její interní typový člen.
- Interní třídy mohou mít deklarován atribut přístupu `private`. Pak o nich žádná z okolních tříd neví.

1.21 Hodnotové a odkazové objektové typy

- Objektové datové typy můžeme rozdělit na hodnotové a odkazové.
- Hodnotové datové typy považují dvě instance za ekvivalentní, pokud obě reprezentují stejnou hodnotu.
- Ekvivalenci dvou instancí porovnává metody `equals(Object)` definovaná ve třídě `Object`.
- Pro implicitní verzi této metody, tj. pro verzi zděděnou od třídy `Object`, je instance ekvivalentní pouze sama se sebou.
- Hodnotové datové typy definují vlastní verzi metody `equals(Object)`, v níž je specifikováno, jak se pozná, že jsou dvě instance ekvivalentní, tj. že reprezentují stejnou hodnotu.
- Kontrakt metody `equals(Object)` vyžaduje, aby metoda byla reflexivní, symetrická, tranzitivní, nenullová, konzistentní a robustní.

- ☞ Hodnotové (objektové) datové typy můžeme ještě rozdělit na proměnné a neměnné.
- ☞ Neměnné datové typy můžeme používat obdobně jako typy primitivní. Můžeme např. bezpečně deklarovat veřejné konstanty těchto typů.
- ☞ Typickými neměnnými hodnotovými datovými typy jsou výčtové typy a třída `String`.
- ☞ Hodnotové datové typy bychom neměli definovat jako proměnné, protože jejich metoda `equals(Object)` nemůže dodržet požadovaný kontrakt.
- ☞ Deklarace veřejných konstant proměnných hodnotových typů je velice nebezpečná.
- ☞ Neměnný hodnotový typ by měl mít definovány své atributy jako konstanty, aby je nebylo možno v průběhu jeho života měnit.
- ☞ Definujeme-li neměnný datový typ, jehož některé metody mají manipulovat s reprezentovanou hodnotou, musíme je definovat tak, že výsledkem každé manipulace bude nová instance daného typu reprezentující upravenou hodnotu.
- ☞ Dceřiná metoda překrývající rodičovskou může deklarovat obecnější typy svých parametrů než její rodičovský ekvivalent a smí vracet specializovanější návratovou hodnotu. Obráceně je to syntaktická chyba.

1.22 Složitější rozšíření funkčnosti

V této kapitole jsme nepřidávali žádné další teoretické poznatky, spíše jsme si ukazovali úpravy funkcionality třídy z praktického hlediska. Pokoušel jsem se vám předvést, jak je výhodné mít předem napsané testy a provádět pak jednotlivé úpravy kódu v malých, snadno kontrolovatelných krocích.

1.23 Budete si to přát zabalit?

- ☞ V zájmu lepší spravovatelnosti bývají programy rozděleny do balíčků.
- ☞ Balíčky tvoří hierarchickou stromovou strukturu obdobně jako soubory na disku.
- ☞ Platforma Java vyžaduje, aby na disku byly class-soubory datových typů uloženy do složek odpovídajících jejich balíčkům. Stromová struktura složek tak bude automaticky odpovídat struktuře balíčků.
- ☞ Rozumní programátoři dodržují shodnou strukturu balíčku a složek i pro ukládání zdrojových souborů. Do jednoho balíčku pak patří třídy, jejichž zdrojové, resp. přeložené soubory jsou ve stejné složce.
- ☞ Každé vývojové prostředí definuje vlastní způsob označení kořenové složky stromu balíčků.
- ☞ Název balíčku sestává z vlastního názvu balíčku, kterému předchází název rodičovského balíčku oddělený tečkou.
- ☞ Podle konvence se pro názvy balíčků používají pouze malá písmena.
- ☞ Příslušnost třídy k balíčku musíme definovat v příkazu `package`.
- ☞ Příkaz `package` musí být úplně prvním příkazem v souboru. Před ním smějí předcházet pouze bílé znaky a/nebo komentáře.

- ☞ Úplný název třídy je dán názvem balíčku následovaným tečkou a vlastním názvem třídy.
- ☞ Chceme-li pracovat s třídami z jiných balíčků, musíme buď používat jejich úplné názvy, nebo musíme jejich názvy nejprve dovézt pomocí příkazu `import`.
- ☞ Před příkazem `import` smí předcházet pouze příkaz `package` nebo jiný příkaz `import`.
- ☞ Příkaz `import` sestává z klíčového slova `import` následovaného úplným názvem dovážené třídy a středníkem.
- ☞ Jeden příkaz `import` nemůže současně importovat dvě třídy. Může však importovat všechny třídy jednoho balíčku.
- ☞ Nahradíme-li v příkazu `import` název dovážené třídy (tj. část úplného názvu za poslední tečkou) hvězdičkou, dovezeme názvy všech tříd z uvedeného balíčku. Používání této hvězdičkové konvence se však nedoporučuje.
- ☞ Podbalíček nepatří do balíčku. Dovezením všech názvů tříd v daném balíčku ještě nedovážíme třídy jeho podbalíčků – ty potřebují vlastní příkaz `import`.
- ☞ Jediný balíček, jehož třídy není třeba importovat, je systémový balíček `java.lang`.
- ☞ V dalších projektech už nebudeme do kořenového balíčku umisťovat žádné třídy.
- ☞ 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. Tato hladina přístupových práv bývá označována jako *package private*.
- ☞ Přístup *package private*, tj. soukromý v rámci balíčku, je v Javě nastaven jako implicitní.
- ☞ Jako soukromé v rámci balíčku je možné definovat i třídy. U takovýchto tříd se v jejich hlavičce neuvede žádný modifikátor přístupu.
- ☞ Třídy, které jsou soukromé v rámci balíčku, je rozumné používat pouze tehdy, bude-li je využívat více tříd z daného balíčku a přitom by neměly být přístupné třídám mimo balíček.
- ☞ Potřebuje-li takovouto třídu využívat pouze jedna třída, bývá výhodnější ji definovat uvnitř třídy, která ji potřebuje, jako její interní datový typ.
- ☞ Zdrojové kódy tříd, které jsou soukromé v rámci balíčku (*package private*), nemusí být uloženy v souboru, který má stejný název jako jméno třídy. Mohou být dokonce uloženy v souboru, ve kterém je již uložen zdrojový kód jiné třídy. Těchto možností však není vhodné využívat.
- ☞ Kořenový balíček je degenerovaný. Datové typy, které jsou v něm definovány, není možno použít v jiných balíčcích. I když budou mít deklarován modifikátor `public`, budou se chovat, jako kdyby měly nastavena přístupová práva *package private*.
- ☞ Bude-li součástí aplikace pouze část stromu balíčků používaných v daném projektu, je vhodné před exportem aplikace do JAR-souboru tuto používanou část stromu balíčků nejprve zkopírovat do nového, prázdného projektu, a teprve tento projekt exportovat do souboru JAR.
- ☞ Vedle klasických příkazů `import` zavedla Java 5.0 také jejich statické verze, které umožní používat statické členy importované třídy bez kvalifikace.
- ☞ Statický import definujeme přidáním klíčového slova `static` mezi slovo `import` a název importovaného objektu.
- ☞ Statický import se doporučuje používat co nejméně.

1.24 Modul × komponenta × knihovna × framework

Knihovny podprogramů se používají téměř 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.

Modul může se sám skládat z podmodulů, které jsou zase vzájemně nezávislé a plní dílčí funkce většího modulu. Definice jazyka Java považuje za ekvivalenty klasických modulů balíčky. V řadě případů jsou jako příklady malých modulů uváděny i třídy (třída je modul obsahující sadu definic metod).

Komponenta

Komponenta je modul, který je možno použít samostatně. Má definované rozhraní a deklaruje požadavky na rozhraní komponent, s nimiž má spolupracovat. Se svým okolím komunikuje pouze prostřednictvím tohoto rozhraní.

Důležitou vlastností komponent je, že jsou nahraditelné jinou komponentou se stejným rozhraním a funkcionalitou, a to nejenom v době návrhu, ale i za běhu programu.

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.

Jako příklad velice jednoduchých komponent jsou někdy uváděny třídy, resp. jejich instance. Instanci třídy můžete (často i za chodu) nahradit instancí 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 určených k použití okolním programem. V dřívějších dobách byly těmito entitami procedury a funkce a později 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 je vlastně knihovna s několika speciálními vlastnostmi.

- ☞ Organizace řešení nespočívá na bedrech uživatele, ale využitím principu inverze závislosti (viz minulý díl) je vše v rukou frameworku, jemuž se musí jeho uživatel přizpůsobit.
- ☞ Framework má definované smysluplné implicitní chování.
- ☞ Framework je rozšiřitelný prostřednictvím cíleného překrytí části kódu kódem definovaným uživatelem.
- ☞ Framework obecně není modifikovatelný. Přesněji řečeno: veškeré požadavky na úpravu či rozšíření jeho funkčnosti by měly být realizovatelné bez zásahu do jeho kódu.

Příkladem frameworku by mohl být správce plátna (instance třídy `CanvasManager`) spolu s instancemi spolupracujících tříd. Chceme-li využívat služeb správce plátna, musíme se mu přizpůsobit tím, že implementujeme rozhraní `IPaintable` a přihlásíme se do jeho správy. Správce pak sám rozhodne, kdy se máme nakreslit. Má rozumné implicitní chování, i když po něm nic nechceme, a v minulém dílu jsme si vyzkoušeli, že vhodnou definicí tříd implementujících požadovaná rozhraní 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.

§