INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

URČENO PRO VZDĚLÁVÁNÍ V AKREDITOVANÝCH STUDIJNÍCH PROGRAMECH

FRANTIŠEK HUŇKA

ČÍSLO OPERAČNÍHO PROGRAMU: CZ.1.07
NÁZEV OPERAČNÍHO PROGRAMU:
VZDĚLÁVÁNÍ PRO KONKURENCESCHOPNOST
OPATŘENÍ: 7.2
ČÍSLO OBLASTI PODPORY: 7.2.2

INOVACE VÝUKY INFORMATICKÝCH PŘEDMĚTŮ VE STUDIJNÍCH PROGRAMECH OSTRAVSKÉ UNIVERZITY

REGISTRAČNÍ ČÍSLO PROJEKTU: CZ.1.07/2.2.00/28.0245

OSTRAVA 2012

Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky

Recenzent: RNDr. Jaroslav Žáček, Ph.D.

Název: Objektově orientované programování
Autor: doc. Ing. František Huňka, CSc.
Vydání: první, 2012
Počet stran: 151

Jazyková korektura nebyla provedena, za jazykovou stránku odpovídá autor.

OBSAH

Úvod pro práci s textem pro distanční studium

Cíl předmětu

Seznámit studenty se základy objektově orientovaného programování. Učební text je zaměřen na vysvětlení základních pojmů a jejich praktickou aplikaci na jednoduchých příkladech. Každý složitější problém je doplněn diagramem tříd pro názornější představu o řešené oblasti. Učební text zahrnuje vykreslení jednoduchých grafických objektů, práci s poli jako datovými atributy, vytváření a práci se zásobníkem, frontou, spojovým seznamem a binárními stromy. Pro názorné pochopení probírané látky je text doplněn několika případovými studiemi.

Po prostudování textu budete znát:

Tento učební text je určen studentům informatiky pro předmět *Objektově orientované programování*. Po prostudování textu a praktickém ověření objektově orientovaného přístupu na příkladech, budete rozumět a umět využívat principy objektově orientovaného programování v praktických aplikacích. Cílem učebního textu je nejen seznámit studenty s teorií objektově orientovaného přístupu, ale také ji umět prakticky aplikovat.

V textu jsou dodržena následující pravidla:

- je specifikován cíl kapitoly (tedy co by měl student po jejím absolvování umět, znát, pochopit),
- výklad učiva,
- důležité pojmy,
- doplňující otázky a úkoly k textu.

Úkoly

Na konci učebního textu jsou uvedeny tři příklady, které během semestru zpracujete a zašlete ke kontrole vyučujícímu. Zadané příklady odpovídají postupně probíranému učivu. Podrobné informace obdržíte na úvodním tutoriálu.

Pokud máte jakékoliv věcné nebo formální připomínky k textu, kontaktujte autora (frantisek.hunka@osu.cz).

Inovace předloženého textu

Učební text prošel celkovou úpravou. Nově byly doplněny kapitoly týkající se práce s jednoduchými grafickými objekty, využití polí jako datových atributů, objektově orientovaný návrh spojových seznamů a

stromů. Zbylé kapitoly byly upraveny s důrazem na praktické uplatnění získaných vědomostí.

1 Základní pojmy

V této kapitole se dozvíte:

- co je to objekt a jakou má strukturu,
- co jsou to přístupové a modifikační metody,
- jakou mají strukturu jednoduché objektově orientované aplikace.

Po jejím prostudování budete schopni:

- lépe rozumět pojmům objekt, třída,
- budete vědět, jak se třída graficky zobrazuje v diagramu tříd UML,
- budete umět vytvořit a spustit jednoduchý objektově orientovaný program.

Klíčová slova této kapitoly:

objekt, instance, datový atribut, třída

1.1 Úvod do objektů a tříd

V objektově orientovaném programování (OOP) se na rozdíl od klasického (procedurálního) programování pracuje pouze s *objekty***.** *Objekt* představuje stěžejní pojem objektově orientovaného přístupu. S tímto pojmem se setkáváme jak v reálném světě, jehož určitou oblast (doménu) realizujeme prostřednictvím vytvářené programové aplikace, tak také ve vlastní programové aplikaci. Tato charakteristická vlastnost OOP do jisté míry usnadňuje chápání a porozumění návrhu implementace objektově orientovaného programu. Je to jedna ze tří výhod OOP, která říká, že s pojmy (objekty) reálného světa pracujeme i při vytváření objektově orientovaného programu. Tedy objektově orientovaný přístup má velmi blízko k reálnému světu, jehož část modelujeme prostřednictvím programové aplikace. Proto v rámci OOP není násilný přechod od reálného světa do světa vytvářené programové aplikace. Používají se stejné pojmy, kterým rozumí jak tvůrce programové aplikace, tak i neškolený zákazník, obeznámený s doménou programová aplikace.

1.1.1Pojem objektu

V objektově orientovaném přístupu je objekt např. osoba, student, kniha, seznam vykonaných zkoušek, seznam dětí, auto atd. Jednotlivé objekty mohou být tvořeny jinými objekty např. auto je složeno

z karoserie, 4 kol, motoru, podvozku, přístrojové desky a objekty můžeme řadit (třídit, klasifikovat) do hierarchických struktur ve smyslu specializace a generalizace. Např. objekt *osoba* je obecnějším (generalizovanějším) objektem než objekty *student* nebo *pracovník* a podobně objekt *nákladní auto* je specializovanějším objektem než objekt *auto*. Objekt z pohledu OOP charakterizujeme jako samostatnou samo identifikovatelnou jednotku, která je navíc zapouzdřena. Pojem zapouzdření hraje důležitou úlohu v OOP a bude vysvětlen poté, co si uvedeme podrobněji, z jakých částí se objekt skládá.

Struktura objektu
Objekt se skládá principiálně ze dvou částí:

- z části, která popisuje vlastnosti objektu (vlastnostem objektu říkáme v OOP **datové atributy**, vnitřní data) – *datová povaha objektu*,
- z části, která popisuje operace (**metody**), které můžeme provádět s datovými atributy objektu (vlastnostmi objektu) – *funkční povaha objektu* .

Datová povaha objektu je dána tím, že objekty se skládají z příslušných vnitřních dat (datových atributů), což jsou buď primitivní typy (real, int, boolean, char), nebo jiné objekty (ze kterých je pak konkrétní objekt složen).

Funkční povaha každého objektu je dána tím, že každý objekt má jakoby kolem svých datových atributů (vnitřních dat) obal či zeď, která je tvořena množinou samostatných částí kódu, jež jsou nazývány ***metodami*** a které realizují požadované **operace**. **Metody** slouží k tomu, aby umožňovaly konkrétní operace s datovými atributy objektu. Metoda má své jméno, deklaruje typy vstupních a výstupních parametrů a obsahuje kód požadovaných dílčích operací.

Zapouzdřenost objektu je důležitá vlastnost OOP a znamená, že s datovými atributy objektu nemůžeme pracovat přímo, ale pouze prostřednictvím deklarovaných **metod** objektu. Každý objekt dovoluje provádět jen ty operace, které povoluje jeho množina metod, které říkáme protokol metod. Proto se hovoří o zapouzdření dat uvnitř objektů. To že k manipulaci s datovými atributy objektu můžeme použít pouze jeho metody je velmi důležitý prvek OOP, který vlastně říká, že s datovými atributy objektu můžeme pracovat pouze způsobem, který je uložen (popsán) v jejich metodách a ne jinak. Pokud skutečně potřebujeme doplnit funkčnost protokolu metod, máme možnost to provést prostřednictvím dědičnost eventuálně využitím vhodných návrhových vzorů.

Množina povolených operací s objektem se nazývá ***protokol metod*** objektu, což je také z hlediska vnějšího systému jeho jediný a plně postačující popis (charakteristika). Popis vnitřní struktury objektu

(datových atributů) není vzhledem k jejich zapouzdření a závislosti na metodách z hlediska vnějšího systému důležitý.

Objekt je **určitá jednotka**, která modeluje nějakou část reálného světa a z funkčního pohledu víc odpovídá *„malému kompaktnímu programu“*, než jedné proměnné příslušného datového typu. Výsledná objektově orientovaná aplikace je potom souborem takových vzájemně interagujících (komunikujících a ovlivňujících se) malých programových celků (objektů).

Přístupové a modifikační metody (metody get/set)
Mezi základní metody, které se deklarují u každého objektu, jsou tzv. *přístupové* a *modifikační* metody. V jazyce Java jsou tyto metody označovány jako metody *get/set* (**get** – zpřístupní, **set** – nastaví, modifikuje). Pokud platí pravidla zapouzdření (využívají se k tomu mj. modifikátory *private*, *public*, *protected*), pak se k žádnému datovému atributu nemůžeme dostat přímo, ale musíme použít odpovídající metodu. Pouze uvnitř třídy je možné přistupovat k datovým atributům přímo, ale i tam se budeme snažit pracovat s využitím metod, protože to má své výhody.

Abychom se tedy dostali ke konkrétní hodnotě primitivního datového typu, nebo k odkazu na objektový typ, musíme použít konkrétní přístupovou metodu (get() např.getVek()). Přístupová metoda deklaruje pouze typ výstupního parametru. Výsledkem této metody je pak konkrétní hodnota primitivního datového typu, nebo odkaz na datový atribut, který je deklarovaný jako objekt. Podobně pokud chceme změnit hodnotu primitivního datového typu (set()), nebo měnit odkaz objektového typu, použijeme k tomu modifikační metodu. Jako vstupní parametr musí být deklarovaná nová hodnota nebo nový odkaz. Abychom se při větších aplikacích nemuseli zabývat těmito jednoduchými metodami, poskytují některá vývojová prostředí automatické vytváření (generování) přístupových a modifikačních metod.

1.1.2 Pojem třídy
Většinou, když vytváříme objekty, potřebujeme vytvořit více objektů dané struktury. Tyto objekty potřebujeme vytvořit podle nějaké „šablony“ (formy, předpisu). Tuto šablonu představuje třída. Třídu si můžeme představit jako „továrnu“ na objekty. Pojem třída (anglicky class) je klíčové slovo objektově orientovaných jazycích. Třídu vytvoříme jedenkrát a pak od ní vytváříme *objekty*, někdy se spíše místo slova objekt používá slovo *instance*. V tomto učebním textu budeme oba pojmy tedy *objekt* a *instance* považovat za rovnocenné (identické). Struktura třídy je následující:

```
public class Trida {
   // deklarace datovych atributu
      . . .
   // deklarace metod
   . . .
}
```

Vidíme, že deklarace třídy je uzavřena v bloku, který tvoří dvojice složených závorek. Většinou volíme takový postup, že deklarace třídy se uloží do jednoho souboru, který má stejný název jako obsažená třída. Např. třída *Bod* se uloží do souboru *Bod.java*. Instance (objekty) od dané třídy pak vytváříme v jiném souboru, který můžeme nazvat *BodTest.java* v tzv. hlavní metodě. Nyní si ukážeme jednoduchý příklad „Hello world“ v objektově orientovaném pojetí. Třída *Hello* je uložena v souboru *Hello.java* a třída *HelloTest* je uložena v souboru *HelloTest.java*.

```
public class Hello {
   // datovy atribut
   private String pozdrav = "Hello World";

   // metoda
   public void go() {
         System.out.println("\n" + pozdrav);
   }
}
```

```
1 public class HelloTest {
2   public static void main(String[] args) {
3      // vytvoreni instance (objektu)
4      Hello hello = new Hello();
5      hello.go();
6      }
7 }
```

Třída *Hello* obsahuje datový atribut *pozdrav* a metodu *go()*, která zobrazí obsah datového atributu *pozdrav* na nový řádek. Třída *Hello* tedy slouží k deklaraci konkrétní třídy.

Třída *HelloTest* obsahuje v sobě tzv. hlavní metodu, ve které se vytvářejí konkrétní instance (objekty) tříd a jednotlivým instancím se zasílají zprávy, na které instance reagují provedením metody, která odpovídá zaslané zprávě. Proměnná *hello* ukazuje na nově vytvořenou instanci třídy *Hello* – řádek 4. Na řádku 5. se zasílá zpráva *go* instanci *hello*, což má za následek vyvolání metody *go()* – která provede zobrazení pozdravu.

Další příklad, který uvedeme, se týká vytvoření třídy, která bude mít jeden celočíselný datový atribut *pocet* a její metody budou umožňovat zvýšení, snížení o 1, nulování a tisk tohoto datového atributu. Třída se nazývá *Citac* a třída s hlavní metodou *CitacTest*.

Třída Citac

```
public class Citac {
   // datovy atribut
   private int pocet;

   // metody
   // pristupova metoda k atributu pocet
   public int getPocet(){
      return pocet;
   }

   public void pricti(){
      pocet = pocet + 1;
   }

   public void odecti(){
      pocet--;
   }

   public void nuluj(){
      pocet = 0;
   }

   public String toString(){
      return "Citac stav: " + getPocet();
   }

   public void tisk(){
   //this - odkaz na sebe sama
        System.out.println(this.toString());
   }
}
```

Třída CitacTest

```
1 public class CitacTest {
2   public static void main(String[] args) {
3      Citac ales = new Citac();
4      Citac eliska = new Citac();
5      ales.odecti();
6      ales.odecti();
7      eliska.pricti();
8      eliska.pricti();
9      eliska.odecti();
10     System.out.println("Tisk citace ales: ");
11     ales.tisk();
12     System.out.println("Tisk citace eliska: ");
13      eliska.tisk();
       }
}
```

V hlavní metodě třídy *CitacTest* jsme vytvořili dvě instance třídy *Citac*, na které odkazují proměnné *ales* a *eliska* (řádek 3 a 4). Instancím zasíláme zprávy *odecti* a *pricti*, které vyvolávají odpovídající metody *odecti()* a *pricti()*.

Za pozornost také stojí, že obě vytvořené instance třídy *Citac*, na které odkazují proměnné *ales* a *eliska* mají své vlastní hodnoty čítače, které, jak je vidět z příkladu, se liší. To znamená, že v deklaraci třídy uvádíme datové atributy a každá instance pak má své hodnoty datových atributů „pro sebe“. Tedy každá instance má pro své datové atributy oddělené místo v operační paměti.

Když si do třídy *Citac* přidáme kromě přístupové metody (get) také modifikační metodu (set), můžeme upravit metody *pricti(), odecti(), nuluj(), toString()* tak, aby využívaly pouze přístupové a modifikační metody. V těchto metodách se tedy nebudeme odkazovat přímo na datový atribut *pocet*, ale budeme tak činit pouze prostřednictvím přístupových a modifikačních metod. Pokud se datový atribut *pocet* přejmenuje, budou tím ovlivněny pouze jeho přístupové a modifikační metody. Ostatní metody zůstanou beze změny.

Třída Citac - rozšíření

```
public class Citac {
   // datovy atribut
   private int pocet;

   // pristupova metoda k atributu pocet
    public int getPocet(){
       return pocet;
       }

    // modifikacni metoda atributu pocet
    public void setPocet(int cislo) {
       pocet = cislo;
    }

    public void pricti(){
       //pocet = pocet + 1;
       this.setPocet(this.getPocet() + 1);
    }

    public void odecti(){
       setPocet(getPocet() - 1); //pocet--;
    }

    public void nuluj(){
       this.setPocet(0); //pocet = 0;
    }

    public void prictiCislo(int cislo) {
       setPocet(getPocet() + cislo);
    }
```

```
    public String toString(){
       return "Citac stav: " + getPocet();
    }

    public void tisk(){
       //this - odkaz na sebe sama
       System.out.println(this.toString());
    }
}
```

Kromě úpravy metod jsme také přidali metodu *pricti(číslo)*, která modifikuje relativně stav čítače o hodnotu parametru číslo (je-li kladná, přičte se, je-li záporná, odečte se).

Třída Citac – další úpravy

```
public class Citac {
    // datovy atribut
    private int pocet;

    // pristupova metoda k atributu pocet
    public int getPocet() {
       return pocet;
    }

    // modifikacni metoda atributu pocet
    public void setPocet(int cislo) {
       pocet = cislo;
    }

    public void pricti(){
       //pocet = pocet + 1;
       this.setPocet(this.getPocet() + 1);
    }

    public void odecti(){
       setPocet(getPocet() - 1); //pocet--;
    }

    public void nuluj(){
       this.setPocet(0); //pocet = 0;
    }

    public void prictiCislo(int cislo) {
       setPocet(getPocet() + cislo);
    }

    public String toString(){
       return "Citac stav: " + getPocet();
    }

    public void tisk(){
       //this - odkaz na sebe sama
       System.out.println(this.toString());
    }
}
```

V kódu předchozího programu se vyskytuje „pseudoproměnná“ *this*. V komentáři k ní je uvedeno, že odkazuje na sebe. *this* zastupuje v tomto případě třídu, ve které je *this* uvedeno. Jak se ve druhé kapitole dozvíte, při posílání zpráv je třeba uvést příjemce (objekt, instance) tečku a zprávu, která vyvolá odpovídající metodu. Jedná-li se o danou třídu, uvádíme příjemce jako *this*. Pokud ale napíšeme pouze zprávu bez příjemce, jako je to např. v metodě *toString()*, která obsahuje pouze zprávu *getPocet()*, pak překladač vždy místo chybějícího příjemce doplní pseudoproměnnou *this*.

Zkuste vytvořit třídu *Citac*, která bude mít navíc datový atribut jméno typu String. Je třeba doplnit přístupovou metodu *getJmeno()* a modifikační metodu *setJmeno(noveJmeno)* a také upravit metodu *toString()*.

Metoda toString()

Každá třída v Javě je implicitně podtřídou třídy *Object*, která je nejvyšší třídou v hierarchii tříd. Třída *Object* deklaruje metodu *toString()*, která vypisuje textovou reprezentaci dané třídy. V podstatě se jedná o výpis hodnot všech datových atributů. Je třeba, aby každá třída deklarovala tuto metodu a tím způsobí zastínění (předeklarování, overriding) metody *toString()* ve třídě *Object*. Pokud to třída neprovede, bude se vyvolávat metoda *toString()* třídy *Object* a ta zobrazí neaktuální (nepravdivé) informace.

Třída CitacTest

```
public class CitacTest {
   public static void main(String[] args) {
       Citac ales = new Citac();
       Citac eliska = new Citac();
       ales.setPocet(11);
       eliska.setPocet(3);
       ales.prictiCislo(15);
       ales.odecti();
       eliska.prictiCislo(-24);
       eliska.pricti();
       System.out.println("Tisk citace ales: ");
       ales.tisk();
       System.out.println("Tisk citace eliska: ");
      eliska.tisk();
       }
}
```

1.2 Výhody objektově orientovaného přístupu

Výhody objektově orientovaného přístupu jsou především tyto:

- blízkost chápání reálného světa,
- stabilita návrhu,
- znuvupoužitelnost.

1.2.1 Blízkost chápání reálného světa

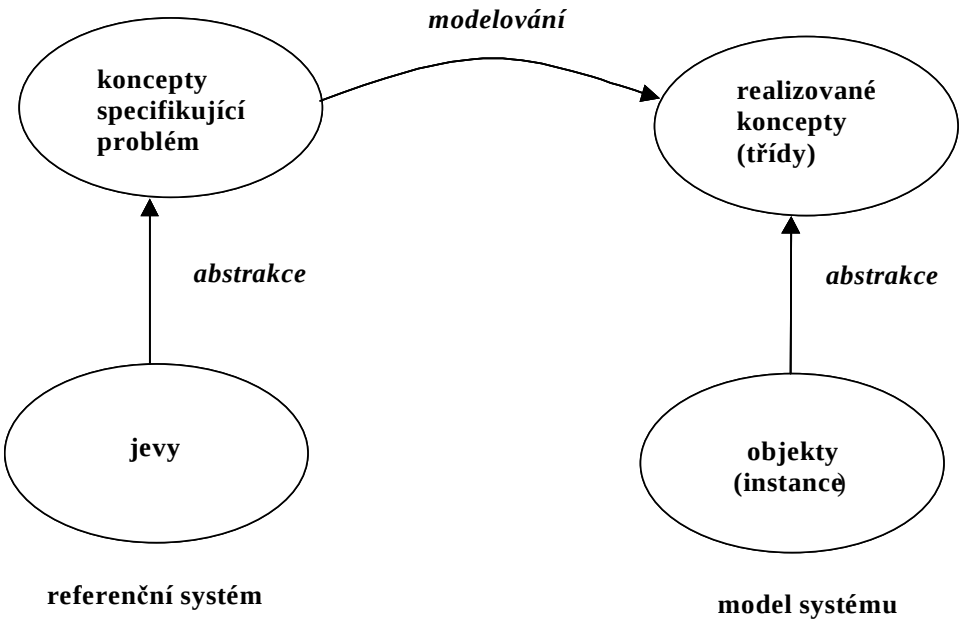
V objektově orientovaném přístupu se vyjadřujeme a pracujeme v pojmech reálného světa a ty se neliší od způsobu chápání reálného světa. Objektově orientovaná analýza je založena na pojmech, které jsme se nejprve učili ve školce; objekty, atributy, třídy, celky a části. Programování je chápáno jako proces modelování viz obr. 1.1. Na levé straně obrázku je referenční systém, který tvoří doménu, kterou modelujeme a na pravé straně je počítačový systém – počítačová reprezentace systému reálného světa.

Systém reálného světa (referenční systém) je složen z **jevů** (jednotlivin) reálného světa. Tyto jevy, jednotliviny představují konkrétní věci reálného světa. Např. tato tužka, tamto stojící zelené auto, modré tričko, které mám na sobě atd. Pomocí jevů se vyjadřují ty nejmenší děti.

Koncepty se získají zobecněním (abstrakcí) **jevů** (jednotlivin). Například když řekneme tužka, nemáme na mysli konkrétní tužku, ale objekt splňující dané vlastnosti. Stejně tak auto jako koncept pro nás může představovat nějaké osobní auto (třeba i dané značky) reprezentující dané vlastnosti.

Jev je věc, která má danou individuální existenci v reálném světě, nebo v mysli; cokoli reálného.

Koncept je zevšeobecňující představa kolekce jevů, založena na znalostech společných vlastností jevů v kolekci.

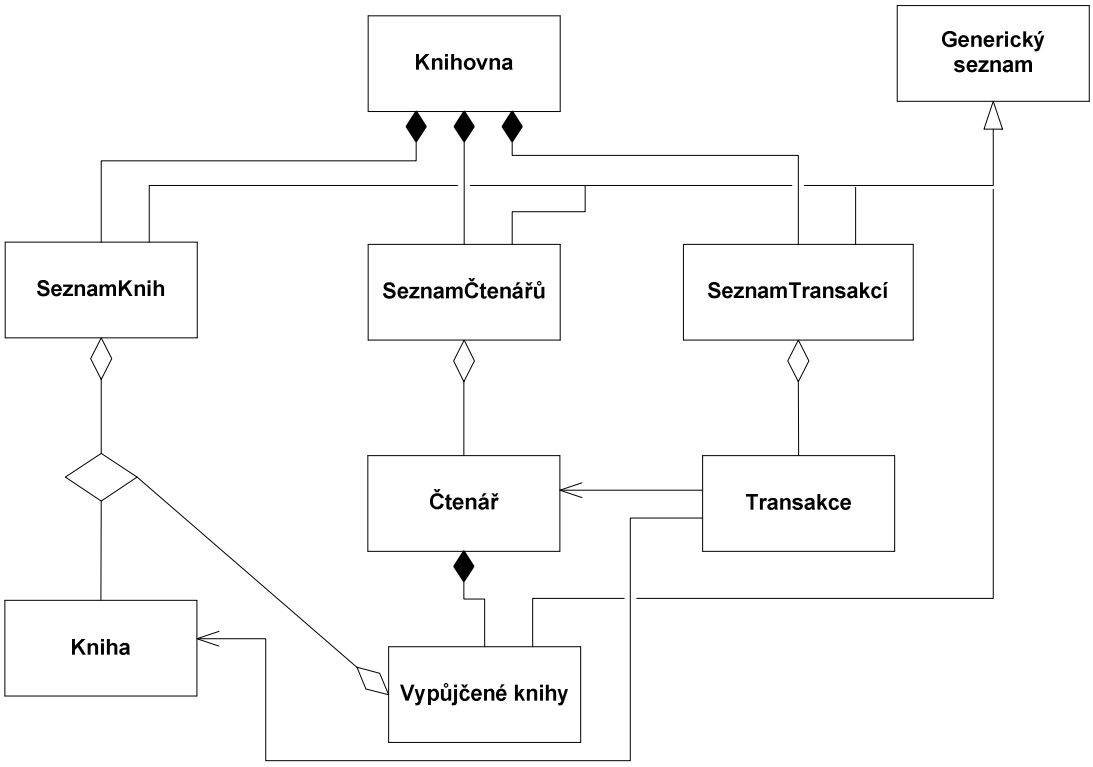


Obr. 1.1 Programování jako proces modelováni

Na úrovni modelu systému musí existovat odpovídající prvky, aby se reálný systém mohl být modelován pomocí počítačového modelu. Tyto prvky jsou představovány objekty (někdy se používá také označení instance). **Jevy** reálného světa jsou modelovány do **objektů** počítačového světa. Abstrakcí těchto objektů dostáváme tzv. **realizované koncepty**, kterým se v počítačové terminologii říká **třídy**. Třída reprezentuje množinu objektů stejného typu. Jak je z obrázku názorně vidět, **koncepty** reálného světa jsou modelovány do **tříd** modelu systému.

1.2.2 Stabilita návrhu

Místo zaměření se na funkcionalitu systému, je prvním krokem vytvoření fyzikálního modelu reálného světa odpovídající dané aplikaci. Tento model je tvořen množinou tříd, které mají mezi sebou jasně specifikované vazby (asociace, kompozice, agregace, dědičnosti). Tento model potom vytváří základ pro různé funkce, které systém může mít. Tyto funkce mohou být později měněny a mohou být dodávány nové funkce beze změny základního modelu, tedy struktury tříd a jejich vazeb. Objektově orientované programování poskytuje přirozenou kostru pro modelování aplikační domény.



Obr. 1.2 Jednoduchý diagram tříd knihovny

Z diagramu na obr. 1.2 je vidět, že knihovna se skládá ze seznamu čtenářů, seznamu knih a seznamu transakcí. Transakce zaznamenává všechny pohyby v knihovně, tedy vypůjčení knihy nebo vrácení knihy. Všechny seznamy v knihovně jsou podtřídami generického seznamu, který představuje javovskou knihovní třídu, která poskytuje základní datové struktury a funkčnost pro práci se seznamem. Tento diagram tříd představuje model (kostru) aplikace, která se může modifikovat přidáváním datových atributů nebo metod ke stávajícím třídám nebo k dodatečně vytvořeným podtřídám. Vazby (asociace) mezi třídami zůstávají zachovány.

1.2.3 Znovupoužitelnost

Jedním z dobře známých problémů tvorby software je schopnost znovu použitelnosti programových komponent, když se vytváří nové komponenty. Komponenty v objektově orientovaném přístupu chápeme jako **třídy**. Každá objektově orientovaná platforma poskytuje knihovnu tříd. V knihovně tříd nalezneme hierarchicky řazené třídy a u každé třídy pak popis jejich datových atributů a hlaviček jejich metod. Hlavičky metod představují název metody, vstupní parametry metod a jejich typy a typy výstupních parametrů. Z tohoto popisu můžeme zjistit, že některá třída např. plně vyhovuje našim požadavkům. Pak stačí třídu „importovat“ a vytvořit od ní instanci a tu pak využívat. Druhá možnost znuvupoužitelnosti spočívá v tom, že konkrétní třída nám může principiálně (v podstatě) vyhovovat, ale potřebujeme ještě doplnit nějakou funkčnost eventuálně další datové atributy. V tom

případě vytvoříme od dané třídy podtřídu a v ní dodeklarujeme požadované datové atributy eventuálně požadovanou funkčnost. Pak stačí stejně jako v prvním případě pouze vytvořit instanci od námi vytvořené podtřídy a tu pak použít.

1.3 Vývojové prostředí

Protože nedílnou součástí tohoto kurzu je jazyk UML (viz dále), pro tvorbu aplikací používáme vývojové prostředí BlueJ. Toto prostředí je výhodné zejména pro výuku objektově orientovaného programování a je ke stažení na adrese http://www.bluej.org. Mezi jeho charakteristiky patří:

- jednoduchost,
- názornost – slučuje možnost klasického textového zápisu programu s možností definice jeho architektury v grafickém prostředí. Grafické prostředí splňuje požadavky diagramu tříd jazyka UML. Toto prostředí je schopno vytvořit na základě grafického návrhu kostru programu a průběžně zanášet změny v programu do jeho grafické podoby a naopak změny v grafickém návrhu zanášet do textové podoby.
- interaktivnost – umožňuje přímou práci s objekty.

Ve vývojových prostředích se pracuje s projekty, které od samostatných programů mohou obsahovat jeden, nebo více programů (podle požadavků programátora).

1.4 Jednotný modelovací jazyk (Unified Modeling Language)

Unified Modeling Language – (UML) je standardní jazyk pro specifikaci, zobrazení (vizualizaci) vytváření a dokumentaci programových systémů, stejně také pro business modeling a jiné neprogramové systémy.

UML je nejrozšířenější schéma grafické reprezentace pro modelování objektově orientovaných systémů.

UML reprezentuje kolekci nejlepších inženýrských zkušeností, jaké byly úspěšně prověřeny v modelování rozsáhlých složitých systémů.

UML využívá hlavně grafickou notaci pro vyjádření návrhu programových projektů.

Používání UML pomáhá projektovým týmům komunikovat, zkoumat potenciální návrhy a prověřovat návrh architektury programovým systémům.

Cíle UML

1. Poskytnout uživatelům jednoduchý, expresivní vizuální modelovací jazyk, aby mohly vyvíjet a měnit smysluplné modely.
2. Poskytnout mechanismus na rozšíření a další specifikaci základních konceptů.
3. Být nezávislý na konkrétním programovacím jazyku a vytvářených procesech.
4. Poskytnout formální základ pro porozumění jazyku modelování.
5. Podpořit vysoce úrovňové rozvojové koncepty jako spolupráce, programové balíčky (frameworks), vzory (patterns) a komponenty.
6. Integrovat nejlepší zkušenosti.

Typy diagramů UML

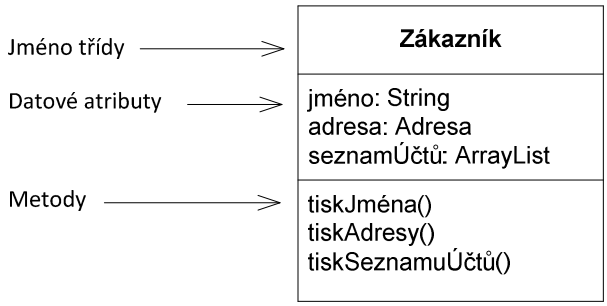
Každý UML diagram je navržen, aby dovolil vývojářům a zákazníkům mít pohled na programový systém z různých perspektiv a z měnících se stupňů abstrakce. UML diagramy obecně vytváření vizuální modelovací prostředky, které zahrnují:

- **Diagram tříd – class diagram**
- Diagram případů užití – use case diagram
- Interakční diagramy
    - Sekvenční diagram
    - Diagram spolupráce
- Stavový diagram
- Diagram aktivit
- Fyzické diagramy
    - Diagram komponent
    - Diagram rozmístění – deployment diagram

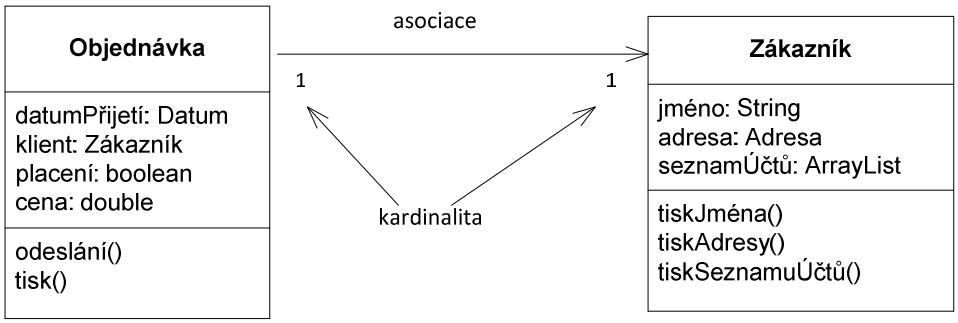
Postupně si vysvětlíme i ostatní diagramy, ale nyní se blíže podíváme pouze na diagram tříd a na jeho nejzákladnější části. Dále se k němu vrátíme v dalších kapitolách.

Diagram tříd

Class diagram (diagram tříd) modeluje strukturu a obsah tříd k čemuž používá navržené prvky jako – třídy, pakety a objekty. Také zobrazuje vztahy (relace) jako kompozice, dědičnost, asociaci a další.



Obr. 1.3 Základní grafická struktura třídy



Obr. 1.4 Grafické označení vazeb mezi třídami

Případ použití (Use Case diagram)

Případ užití je specifikace posloupnosti činností, které systém může vykonávat prostřednictvím interakce (vzájemného působení) s vnějšími účastníky. Používá se hlavně pro vyjasnění požadavků mezi klientem a návrhářem systému. Produkuje jednotlivé scénáře řešení.

Sekvenční diagram

Sekvenční diagram – zobrazuje dynamickou stránku problému, časovou sekvenci objektů účastnící se interakce. Skládá se z vertikální dimenze (čas) a horizontální dimenze (různé objekty).

Diagram spolupráce

Diagram spolupráce zobrazuje interakce organizované mezi objekty a jejich spojení (vazby) mezi sebou (pořadí zasílaných zpráv mezi objekty).

Stavový diagram

Zobrazuje sekvence stavů, kterými prochází objekt během svého života v závislosti na obdrženém stimulu, spolu s jeho reakcemi a činnostmi.

Diagram aktivit

Zobrazuje speciální stavový diagram, kde většina ze stavů jsou stavy činností a většina přechodů je spouštěna vykonáním akcí ve zdrojových stavech. Tento diagram se zaměřuje na toky řízené vnitřním

zpracováním. Používá se k zachycení algoritmů v programovacích jazycích – vývojový diagram.

Diagram komponent

Zobrazuje vysokou úroveň paketové struktury samotného kódu.

Diagram nasazení

Diagram nasazení – zobrazuje konfiguraci prvků běhového zpracování (run-time processing elements) a programových komponent, procesů a na nich žijících objektů. Využívá se v aplikacích pro distribuované prostředí.

V tomto modulu jsou vysvětleny základní pojmy objektově orientovaného programování a názorně ukázány jednoduché příklady využívající objektově orientovaný přístup. Je zmíněn jazyk UML se zaměřením na deklaraci třídy a základních asociací, znázornění kardinalit a kvantifikátorů. V následujících kapitolách budeme základní pojmy dále rozvíjet a praktické aplikace doplňovat o odpovídající diagramy jazyka UML.

Co představuje datovou povahu a co funkční povahu objektu?
Co je grafický symbol pro třídu a její datovou a funkční část?

2. Třídně instanční model

V této kapitole se dozvíte:

- co představuje třídně instanční model,
- jaký je význam zapouzdřenosti dat,
- jak objekty mezi sebou komunikují, jaký je význam zprávy a jaký je význam metody v této komunikaci,
- k čemu a jak se používají konstruktory.

Po jejím prostudování budete schopni:

- budete rozumět komunikaci mezi objekty,
- vytvářet požadované instance pomocí konstruktorů,
- deklarovat a vytvářet nové objekty v metodách, jejichž datové atributy obsahují výsledek metody.

Klíčová slova této kapitoly:

třída, objekt, instance, zpráva, metoda

2.1 Třída

Třída popisuje implementaci množiny objektů, které všechny reprezentují stejný druh systémové komponenty (složky). Třídy se zavádějí především z důvodů *datové abstrakce, znalostní abstrakce, efektivnímu sdílení kódu* a *zavedení taxonomie do popisu programové aplikace*.

V systémech se třídami jsou *objekty* chápány jako *instance tříd*. Třída *OsobníAuto* má pak např. instance *Fabia, Seat, Audi*.

Třída popisuje formu soukromých pamětí a popisuje, jak se provádějí operace. Např. konkrétní třída popisuje implementace objektů, jenž reprezentují obdélníkové plochy. Tato třída popisuje, jak si individuální *instance* pamatují umístění svých ploch a také jak provádějí operace pro obdélníkové plochy.

Programování pak sestává z vytváření tříd a specifikuje sekvenci zpráv, které se vyměňují mezi objekty.

Objekt (instance třídy) odpovídá v reálném světě jevu, třída má svůj protějšek v konceptu. Objekty umí reagovat na zprávy provedením příslušných operací, které jsou popsány ve třídě a sdíleny všemi jejími instancemi. Takové sdílení vede k vyšší efektivnosti při implementaci objektových systémů.

Vztah *třída - instance* můžeme charakterizovat jako vztah *popisu* a *realizace*.

Rozdělení objektů na třídy a instance však mění model výpočtu, protože instance obsahují jen data a metody jsou uloženy mimo ně v jejich třídě. Je-li tedy instanci poslána zpráva, tak instance musí požádat svoji třídu o vydání příslušné metody. Kód metody je poté pouze *dočasně* poskytnut instanci k provedení.

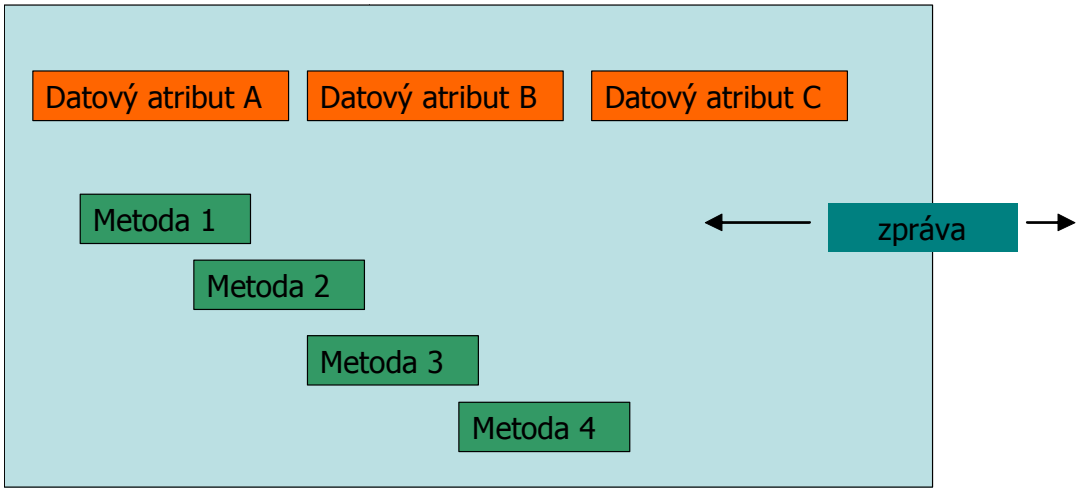
2.2 Objekt (instance)

Základem objektových systémů je objekt. Je to nedělitelná sebe identifikovatelná entita, obsahující datové atributy, jejich identifikaci a metody, které realizují příslušné operace na těchto datech.

Příklady objektů:

- Čísla, řetězce znaků,
- Datové struktury: zásobník, fronta seznam, slovník,
- Grafické obrazce, adresáře souborů, soubory,
- Kompilátory, výpočetní procesy
- Grafické pohledy na informace, finanční historie atd.

Důležitou vlastností objektu je také jeho zapouzdřenost.



Obr. 2.1 Struktura objektu

Zapouzdřenost – encapsulation

Podstatou zapouzdřenosti je, že k datovým atributům přistupujeme **výhradně prostřednictvím metod**. Metody jednoznačně určují, co je dovoleno. Pokud námi požadovaná operace není v žádné metodě, pak máme smůlu a musíme věc řešit jiným způsobem. Zapouzdřenost patří k základním charakteristikám objektově orientovaného přístupu. Její výhody spočívají zejména v:

- Programy mohou být testovány po menších částech.
- Odstraní se častá chyba, která bývá skryta ve sdíleném přístupu ke společným datům.
- Interní datová struktura může být změněna bez nutnosti změn okolí objektu (změna názvu datových atributů).
- Mohou být vytvářeny knihovny objektů, tedy abstrakcí datových typů, které je možné použít v jiných aplikacích.
- Je zabezpečena ochrana a identifikace, se kterými program pracuje.
- Zapouzdření napomáhá k oddělení rozhraní (interface) – viditelná část objektu od implementace (skrytá část deklarace objektu).

Z definice objektu vyplývá, že je pro něj typické spojení dat a operací v jeden nedělitelný celek s ochranou dat, tedy zapouzdřením. **Data jsou spojena s operacemi tak těsně, že se k nim bez těchto operací nedostaneme**.

Sebeidentifikace znamená, že objekt sám o sobě ví kdo je, takže paměť obsahující objekty obsahuje i informace o struktuře svého obsahu.

Ukrývání informací

Při *externím* ukrývání informací máme na mysli to, že objekty by neměly zpřístupňovat přímo svá lokální data a ani kódy jednotlivých operací. Objekt je tedy zvenku neprůhledná entita.
Při *interním* ukrývání informací máme na mysli to, že při dědění nemusí mít následníci objektu přístup k lokálním datům předchůdců a také nemusí mít přístup ke kódu jednotlivých operací svých předchůdců.
Objekty chápeme jako dynamické entity, které v průběhu výpočtu vznikají, vytvářejí nové objekty a zase zanikají.

Objekt (instance) versus odkaz (reference) na objekt

Proměnná v Javě nikdy neobsahuje vytvořenou instanci (objekt), ale pouze odkaz (referenci) na vytvořenou instanci (objekt). Instance je zřízena někde ve zvláštní paměti v haldě (heap). O haldu se stará správce paměti (garbage collector). Jeho funkce:

- přidělování paměti nově vznikajícím objektům,
- rušení objektů, které nikdo nepotřebuje, (na které nejsou žádné odkazy).

Na jednu instanci pak samozřejmě může odkazovat více proměnných. Zrušení instance představuje zrušení všech odkazů na něj.

2.3 Zprávy a metody

Objekty reagují na zprávy (požadavky), které jsou jim v čase adresované. Reakce objektů na zprávu může být:

- jednoduchá odpověď objektu (vrácení hodnoty, nebo objektu),
- změna vnitřního stavu objektu,
- odeslání zprávy jinému objektu,
- vytvoření nového objektu,
- kombinace uvedených možností.

Formalizace zápisu, kdy zpráva zaslaná příjemci nezasílá žádnou odpověď (kvalifikátor void při popisu metody):

```
příjemce.zprava(eventuální parametry zprávy);
```

Formalizace zápisu, kdy zpráva zaslaná příjemci vrací odpověď (která musí být patřičně kvalifikovaná):

```
odpověď  =  příjemce.zpráva(eventuální parametry zprávy);
```

kde:

příjemce reprezentuje objekt (instanci dané třídy,
zpráva je konkrétní zpráva deklarovaná pro danou třídu (eventuálně její nadtřídy),
eventuální parametry zprávy představují skutečné parametry zprávy, které jsou vyžadovány.

Mechanismus posílání zpráv, který je využit v objektově orientovaném přístupu, je založen na výpočetním modelu klient / server. Odpověď představuje klienta a server je reprezentován *příjemcem zprávy*.

Např. třída *String* je poskytovatelem řady standardních služeb pro zpracování řetězců.
Třída *String* – server, který poskytuje řetězcově orientované služby aplikacím – klientům.

Aplikace, která využívá třídu *String* (její objekty) je klient, který vyžaduje služby serveru vyvoláním odpovídajících metod.
Vyvolání metody je prováděno prostřednictvím mechanismu posílání zpráv.

```
// úvodní kvalifikace a
//inicializace objektu s1
String s1 = “Libovolny textovy retezec”;

// n – odpověď obsahuje délku řetězce s1
int n = s1.length();

// řetězec s1 bude obsahovat pouze malá písmena
s1 = s1.toLowerCase();
```

Objekty (instance) komunikují s jinými objekty pomocí posílání zpráv. Množina zpráv, na kterou objekt reaguje se nazývá protokol (protokol zpráv).

Zpráva je **žádost**, aby objekt provedl jednu ze svých operací. Zpráva **specifikuje** o jakou operaci se jedná, ale **nespecifikuje**, jak by se operace měla provést. Objekt, jemuž je poslána zpráva sám **určuje**, jak provést požadovanou operaci. Na provedení operace je nahlíženo jako na vnitřní schopnost objektu, která může být inicializovaná jedině zasláním zprávy.

Běžící objektově orientovaný program je tvořen soustavou mezi sebou navzájem komunikujících objektů, který je řízen především sledem vnějších událostí z rozhraní programu.

2.4 Konstruktory

Konstruktor je speciální metoda, pro vytváření a inicializaci nových objektů (instancí). Název této metody je totožný s názvem třídy. Např.

```
Bod  b1 = new Bod();  // new klíčové slovo
```

nebo delší zápis:

```
Bod b1;   // kvalifikace proměnné b1
b1 = new Bod( ); // vytvoření a inicializace nového objektu
(instance)
```

Konstruktor vytvoří požadovanou instanci a vrátí odkaz, jehož prostřednictvím se na objekt odkazujeme. Konstruktor **musí mít** každá třída. Pokud třída nemá deklarovaný žádný konstruktor, doplní překladač nejjednodušší konstruktor (bez parametrů) a ten se označuje jako implicitní.

Existuje-li pro třídu alespoň jeden programátorem deklarovaný konstruktor (explicitní), překladač žádný implicitní konstruktor nepřidává. Konstruktory dané třídy se **mohou lišit počtem (typem) parametrů**. Definice několika verzí konstruktorů s různými sadami parametrů se označuje jako přetěžování (overloading) daného konstruktoru. Jednotlivé verze konstruktorů se pak nazývají přetížené.

Kopírovací konstruktor

Kopírovací konstruktor je speciálním typem konstruktoru, který má na vstupu parametr stejného typu jako vlastní konstruktor. Např. konstruktor třídy *Osoba* bude mít na vstupu také objekt typu *Osoba*. Hlavní význam kopírovacího konstruktoru je v tom, že kopíruje obsah datových atributů, který má na vstupu parametr konstruktoru, do datových atributů nově vytvářeného objektu (instance). Kopírovací konstruktor je uveden ve třídě Bod a pak i v další kapitole.

2.5 Primitivní a objektové datové typy

Typ údaje popisuje, „co je daný údaj zač“. V typově orientovaných jazycích mají veškerá data, se kterými program pracuje svůj typ, tedy u každého údaje předem znám typ.

Výhody:

- rychlejší práce
- kompletnější kontrola (robustnost)

Java rozlišuje:

- primitivní datové typy
- objektové datové typy

Primitivní datové typy
Jsou to např. čísla – zabudována hluboko v jazyku, chování pevně dané; na vytvoření není třeba konstruktor tedy posílání žádných zpráv:

Tab. 2.1 Přehled primitivních datových typů v Javě

Typ	Velikost v bitech	Zobrazená hodnota
boolean		true false - implementace závislá na JVM (Java Virtual Machina)
char	16	‘\u0000’ až ‘\uFFFF’ (0 – 65 535)
byte	8	-127 + 128
short	16	- 32 767 +32 768
int	32	-2 147 483 648 +2 147 483 647
long	64	- 9 223372 036 854 775 808 +9 223 372 036 854 775 807
float	32	záporné: -3,40 . . E+38 -1,40 . . e-45 kladné: 1,40 . . e-45 3,40 . . E+38
double	64	záporné: -1,79 . . E + 308 -4,94 . . e - 324 kladné: 4,94 . . e – 324 1,79 . . E + 308

Vrácení hodnot primitivních datových typů

`int getX()` - vrací celé číslo
`double getBalance()` - vrací reálné číslo

Objektové typy

Referenční objektové typy jsou objekty (instance) daných tříd. Jedná se o třídy knihoven a uživatelem definované třídy. Standardní knihovna obsahuje cca 1500 tříd

Třída *String* definuje typ znakových řetězců, posloupnost znaků je chápána jako objekt.

Vrácení referencí objektových typů

K převzetí odkazu musíme mít připravený odkaz (referenci) odpovídajícího typu, (která bude vytvořena v zásobníku odkazů) a do které bude požadovaný odkaz na objekt uložen (viz příklad Osoba – Adresa – Ucet). Výjimkou je objektový typ (třída) String, která se někdy chová i jako primitivní typ.

`Adresa getAdresa()` - vrací odkaz na instanci třídy Adresa
`Bod getBod()` – vrací odkaz na instanci třídy Bod

2.6 Implementace třídy Bod

Třída *Bod* bude mít pouze dva datové atributy reprezentující *x* a *y* souřadnici bodu. Dále má třída deklarované další metody pro práci s datovými atributy.

```
public class Bod {
   private int x;
   private int y;

   // konstruktory
   public Bod() {
       //x = 0; y = 0;
       this(0, 0);
    }

   public Bod(int x, int y) {
       this.x = x;
       this.y = y;
   }

   // kopirovaci konstruktor
   public Bod(Bod bod) {
       x = bod.getX();
       y = bod.getY();
   }

   public int getX() {
       return x;
   }
```

```
    public int getY() {
        return y;
    }

    public void setX(int x) {
        this.x = x;
    }

    public void setY(int y) {
        this.y = y;
    }

    public void posunSouradnic(int dx, int dy) {
         setX(getX() + dx);
         setY(getY() + dy);
    }

    public String toString(){
         return "X: " + getX() + " Y: " + getY();
    }

    public void tisk() {
         System.out.println(toString());
    }

    public int delkaX(Bod bod) {
        return Math.abs(getX() - bod.getX());
    }

    public int delkaY(Bod bod) {
        return Math.abs(getY() - bod.getY());
    }

    public Bod soucet(Bod b) {
        Bod vysledek =
              new Bod(getX() + b.getX(), getY() + b.getY());
        return vysledek;
    }

    public Bod rozdil(Bod b) {
        return new Bod(getX() - b.getX(), getY() - b.getY());
     }
}
```

Všimněte si prosím podrobněji metod *soucet()* a *rozdil()*. V obou metodách se jedná o vytvoření **nové instance**, jejíž datové atributy budou v případě *součtu* tvořeny součtem datových atributů daného objektu s objektem, který metoda přijímá jako vstupní parametr. V případě *rozdílu* se jedná o rozdíl odpovídajících datových atributů.

Třída BodTest

```
1 public class BodTest {
2   public static void main(String[] args) {
3       Bod bodA = new Bod(100,110);
4       Bod bodB = new Bod(230, 400);
5
```

```
6       int delkaX = bodB.delkaX(bodA);
7       int delkaY = bodA.delkaY(bodA);
8       System.out.println("delkaX: " + delkaX + " delkaY: "
                                + delkaY);
9
10      Bod c = bodB.soucet(bodA);
11      Bod d = bodA.rozdil(bodB);
12
13      c.tisk();
14      d.tisk();
15      // kopirovaci konstruktor
16      Bod bodF = new Bod(bodA);
17
18      bodA.posunSouradnic(20, -300);
19      bodB.posunSouradnic(-40, 55);
20      bodA.tisk();
21      bodB.tisk();
   }
}
```

V hlavní metodě pak mimo jiné můžete sledovat použití dříve zmíněných metod *soucet()* a *rozdil()* – řádku 10 a 11. Vidíme, že stačí deklarovat odpovídající proměnnou např. *c* v případě *součtu* a ta pak bude ukazovat na novou instanci vzniklou sečtením odpovídajících souřadnic proměnných *bodB* a *bodA*.

Použití kopírovacího konstruktoru je ukázáno na řádku 16. Funkcí tohoto konstruktoru je, že vytvoří novou instanci dané třídy, do které zkopíruje všechny datové atributy objektu, který je předaný jako parametr. Ve třídě BodTest proměnná *bodF* odkazuje na instanci třídy Bod s datovými atributy x = 100, y = 110. Datové atributy jsou stejné jako u proměnné *bodA*, ale obě proměnné *bodA* a *bodF* odkazují na jiné objekty. Kopírovací konstruktor bývá využíván při skládání objektů typu kompozice (pevná vazba mezi celkem a částmi).

Pojem **třídy** je základním pojmem v tzv. třídně instančním modelu objektově orientovaného přístupu. Třída vlastně představuje „továrnu“ na výrobu objektů.

Objekty mezi sebou komunikují formou zasílána zpráv. Protože je zpráva pouze žádost o provedení operace, je plně v kompetenci příjemce zprávy, aby na zprávu reagovat odpovídající svoji metodou.

Kopírovací konstruktor vytváří novou instanci stejné třídy, do které zkopíruje obsah datových atributů objektu, který převzal jako argument.

Co je to třída?
Jaký je rozdíl mezi instancí a objektem?

Třída popisuje implementaci množiny objektů. Třída představuje deklaraci. Objekty, instance jsou vlastně herci na jevišti“, kteří celou práci provedou, tak jak je to deklarované ve třídách a tak jak jsou jim posílány zprávy. Na jakýkoli běžící objektově orientovaný program se můžeme dívat jako na graf objektů. Uzly v grafu jsou objekty a spojnice mezi uzly jsou reference mezi objekty.

Co jsou to přístupové metody, jak se označují a k čemu se používají?

2.1 Co jsou modifikační metody, jak se označují a k čemu se používají?

2.1 Přístupové metody jsou metody ve třídě, které se používají k zpřístupnění datových atributů dané třídy. Zatímco datové atributy mají u sebe většinou modifikátor *private*, metody u sebe většinou mají modifikátor *public*. Většinou se tyto metody označují slovíčkem *get* které je následované identifikátorem daného datového atributu. Např. *getDelka( ),* - metoda vrací datový atribut delka (return delka;).

2.2 Modifikační metody se používají k modifikaci datových atributů. Většinou se tyto metody označují slovíčkem *set*, následovaný identifikátorem datového atributu a v závorkách je typ a nová hodnota, kterou chceme přiřadit datovému atributu. Např. *setDelka(int p);* kde proměnná *p* obsahuje novou hodnotu.
Přístupové a *modifikační* metody se deklarují proto, abychom s datovými atributy nepracovali přímo, ale pouze prostřednictvím metod, které deklarují požadovaný přístup.

3. Práce s objekty v grafickém prostředí (bod, čára)

V této kapitole se dozvíte:

- jak si představit odkaz na objekt a práci s ním,
- k čemu se používá pseudoproměnná this,
- jaký je význam třídních atributů a třídních metod,
- jaké datové atributy musíme doplnit pro vykreslení bodu.

Po jejím prostudování budete schopni:

- lépe rozumět pojmům objekt, třída, budete vědět, jak se třída graficky zobrazuje v diagramu tříd UML, vytvořit a spustit jednoduchý objektově orientovaný program.

Klíčová slova této kapitoly:

odkaz na objekt, instance, grafický objekt

Proměnné ukazující na objektové typy

V předchozí kapitole jsme si na příkladech ukázali, že návratovou hodnotou z metody může být hodnotový typ (např. datový atribut *x* třídy *Bod*). Nyní si ukážeme, že návratovou hodnotou může být také referenční objektový typ a tedy že dvě proměnné mohou ukazovat na stejný objekt (instanci). Do předchozího příkladu doplníme dvě další metody a to metody *getBod*() a kopírovací konstruktor. Zkrácený zdrojový kód třídy je pak následující:

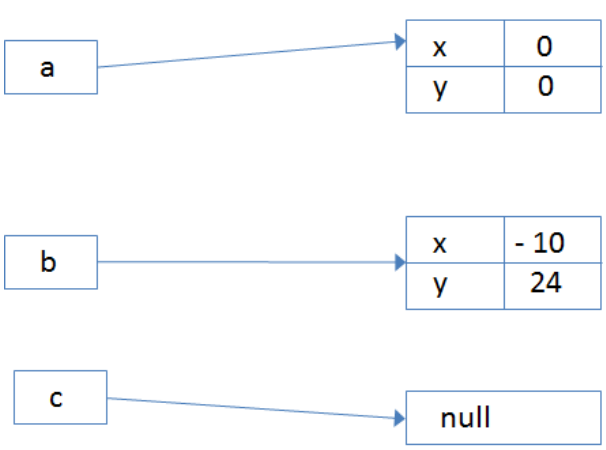
```
public class Bod {
    private int x;
    private int y;

    // . . .

    public Bod(Bod f){
          x = f.x;
          y = f.y; }

    public Bod getBod(){
          return this; }
}
```

Kopírovací konstruktor nastavuje datové atributy příjemce této zprávy na hodnoty datových atributů argumentu *f*. Metoda *getBod*() vrací referenci (odkaz) na příjemce této zprávy. Aby vše bylo názornější ukážeme si použití těchto metod ve třídě *BodTest*:

Třída BodTest

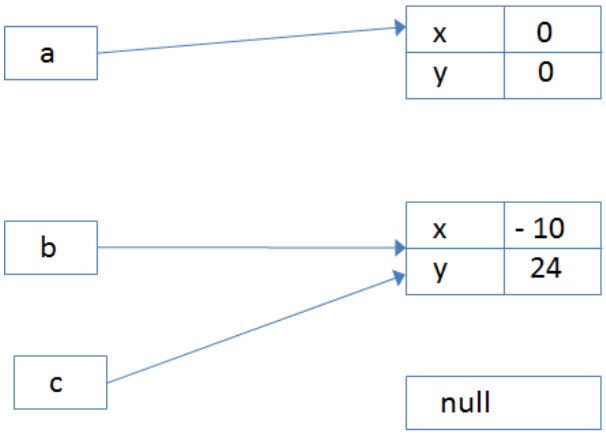
```
public class BodTest {
   public static void main(String[] args) {
    Bod a, b, c; String t;
    a = new Bod();
    b = new Bod(-10, 24);
    if( c == null) System.out.println(“c ukazuje na null”);
    // STOP 1
    c = b.getBod(); // c =  b;
    c.tisk();
    if (b==c) t = "ANO"; else t = "NE";
    System.out.println("Vysledek: "+t);
    // STOP 2
    a = new Bod(c);
    a.tisk();
    b.tisk();
    c.tisk();
   }
}
```

V příkladu jsme použili dva komentáře (STOP 1 a STOP 2). Ty nám kód programu rozdělují na tři části, které si pro větší přehlednost znázorníme graficky. Jedná se nám hlavně o to, aby bylo vidět, že proměnné odkazují na instance (objekty). Když pouze deklarujeme (kvalifikujeme), že proměnná je daného typu, např. ve třídě *BodTest* je deklarovaná proměnná *c* jako proměnná třídy *Bod*, to znamená, že proměnná ukazuje na null.



Obr. 3.1 Situace před komentářem STOP 1

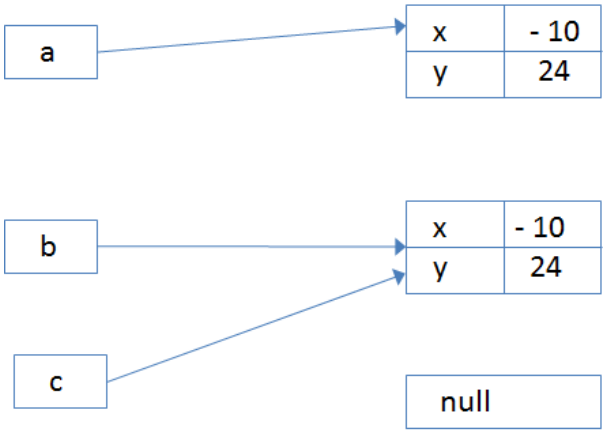
Proměnné *a*, *b* ukazují na nové objekty, jejichž datové atributy jsou nastaveny na odpovídající hodnoty. Proměnná c je pouze kvalifikována na třídu Bod, proto zatím ukazuje na *null*.



Obr. 3.2 Situace mezi komentáři STOP1 a STOP 2

Obrázek 3.2 zobrazuje situaci po provedení příkazu *c = b.getBod()*. Dá se to zapsat jednodušeji *c = b;* Tím proměnná *c* získává odkaz na objekt, na který odkazuje proměnná *b*.

Obrázek 3.3 zobrazuje situaci po provedení příkazu *a = new Bod(c)*. Kopírovací konstruktor způsobí, že datové atributy instance a se nastaví na stejné hodnoty jako datové atributy objektu, na který ukazuje proměnná c. I když instance mají stejné hodnoty svých datových atributů, jedná se o dva různé objekty, jak je patrno z obrázku.



Obr. 3.3 Situace za komentářem STOP 2

3.1 Pseudoproměnná this

Výraz pseudoproměnná, který používáme znamená, že se jedná o proměnnou, ale ne v tom plném slova smyslu. Tato pseudoproměnná má více významů. O prvním významu jsme se již zmínili dříve, kdy jsme řekli, že všude tam, kde napíšeme pouze metodu bez objektu příjemce, doplní překladač jako příjemce pseudoproměnnou this.

Náš zápis:

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

Úprava překladačem:

```
System.out.println(this.toString());
```

Použití pseudoproměnné this je v obou metodách nutné, protože tím rozlišujeme argument (parametr) x, jehož hodnotu předáváme do metody od datového atributu x, který je použitý v deklaraci třídy Bod. V příkazu:

```
public void setX(int x)
       this.x = x;
```

this.x - reprezentuje datový atribut třídy,
x - reprezentuje argument metody.

Pokud bychom použili různé identifikátory, nemusíme pseudoproměnnou this používat. (Pokud ji ale použijeme, nic se nestane, jen se zpřehlední kód programu). Viz následující ukázka metody:

```
public void setX(int k){
        x = k; // this.x = k;
   }
```

3.2 Klíčové slovo final a jeho použití

Někdy je třeba zabezpečit neměnnost datových atributů daných objektů. Máme tím na mysli neměnnost datových atributů, které jsme vytvořili v konstruktoru. Např. počet dnů v týdnu bude stále 7, stejně jako počet haléřů do koruny je 100. To znamená, že chceme přiřadit datovému atributu jednou hodnotu, kterou již nemůžeme změnit.

Také bývá výhodné v programu místo literálů používat pojmenované konstanty. Ty by měly podobný význam jako již zmiňované datové atributy, tedy jednou přiřazená hodnota je dále neměnná.
Pojmenované konstanty se definují jako atributy, - mezi modifikátory se použije klíčové slovo **final**. Např.

```
   private final int TYDEN = 7;
   private final int PRACTYDEN = 5;
```

Z uvedených příkladů vyplývá, že pojmenovaná konstanta TYDEN bude mít vždy hodnotu 7 a pojmenovaná konstanta PRAC_TYDEN hodnotu 5. Podle konvence se pro identifikátory konstant používají velká písmena.

Ukážeme si možnosti využití ve třídě BodFinal.

Třída BodFinal

```
public class BodFinal {
  private final int x;
  private final int y;

  public BodFinal() {
       // initialise instance variables
       x = 0; y=0;
  }

  public BodFinal(int c) {
       x = c; y = c;
  }

  public BodFinal(int x, int y) {
       this.x = x; this.y = y;
  }

  public void setX(int x){
       //this.x = x;
       // can't assign a value to final variable x
  }

  public void setY(int y){
       //this.y = y;
       // can't assign a value to final variable y
  }

  public void setBod(Bod a){
       // x = a.x; can't assign a value to final variable x
       // y = a.y; can't assign a value to final variable y
   }
}
```

Při použití klíčového slova *final* pro datové atributy třídy, překladač umožní pouze inicializaci datových atributů v konstruktoru, avšak další změny nejsou možné. Abychom se zbavili chyb překladače, uvedli jsme tyto metody jako komentář.

3.3 Třídní (statické) atributy, třídní (statické) metody

V nadpisu této části používáme dvě přídavná jména. Statický podle anglického klíčového slova *static*. Třídní znamená působnost na celou třídu, tedy na všechny její instance. To právě vystihuje význam uvedených atributů a metod. Nejdříve se podíváme na třídní atributy:

Doposud jsme deklarovali datové atributy třídy jako tzv. instanční proměnné. Význam těchto atributů je v tom, že v deklaraci třídy nadeklarujeme jejich typy a pak každá instance si naplní tyto datové atributy svými hodnotami, odpovídajícími odkazy na objekty.
Naproti tomu statické nebo třídní atributy jsou atributy deklarované ve třídě, jejichž hodnoty jsou **stejné** pro všechny instance dané třídy. Proto si do nich můžeme uložit nějaké důležité hodnoty např. *krok*

posunu, *barvu*, zvolenou *stupnici* pro měření teploty (Fahrenheit, Celsius, Kelvin) atd.

Instanční proměnné jsou datové atributy objektu. Jejich hodnoty, reference se většinou liší.
Třídní proměnné jsou proměnné třídy. Jejich hodnoty stejné pro všechny objekty (instance) dané třídy.
Klíčové slovo *static* je v deklaraci uvedeno před datovým / objektovým typem.
Notace zápisu:

```
   private static int cislo;
   private static String nazev;
   private static boolean Q;
```

Praktické využití si můžeme ukázat na příkladu třídy *OsobaStat*, kde jako statické atributy jsou deklarované *charita* a *srazka*. *Charita* představuje částku, kterou vyberou všechny instance třídy *OsobaStat* a *srazka* je reálné číslo, které přestavuje poměr, který se z vybrané částky srazí např. formou daní (jedná se pouze o ilustrativní příklad).

Třída OsobaStat

```
public class OsobaStat {
   private String jmeno;
   private static double charita = 1000;
   private static double srazka = 0;

   public OsobaStat(String jmeno) {
      this.jmeno = jmeno;
   }

   public void vyber(double prispevek) {
      charita += prispevek;
   }

   public static double getCharitaCelkem() {
      return charita;
   }

   public static double getCharita() {
      return getCharitaCelkem() –
                  (getCharitaCelkem() * srazka);
   }

   public static void setSrazka(double hodnota) {
      srazka = hodnota;
   }
}
```

Třídní (statická) metoda *getCharitaCelkem()* vrací celkem vybranou částku. Další třídní metoda *getCharita()* vrací čistou částku získanou z vybírání peněz a poslední třídní metoda umožňuje měnit (modifikovat) hodnotu, která bude z vybrané částky sražena.

Třída Osoba StatTest

```
1 public class OsobaStatTest {
2   public static void main(String[] args) {
3     // nastaveni tridni(staticke promenne)
4     OsobaStat.setSrazka(0.2);
5     // tvorba instanci
6     OsobaStat jana = new OsobaStat("Jana");
7     OsobaStat pavel = new OsobaStat("Pavel");
8     OsobaStat eliska = new OsobaStat("Eliska");
9     jana.vyber(200);
10    pavel.vyber(400);
11    eliska.vyber(800);
12    double celkem = OsobaStat.getCharitaCelkem();
13    double cisteho = OsobaStat.getCharita();
14
15    System.out.printf("Celkem vybrano: %.2f, cisteho: %.2f",
16                          celkem, cisteho);
17    OsobaStat.setSrazka(0.25);
18    // . . .
      }
}
```

Třída *OsobaStatTest* ilustruje použití třídních atributů a třídních metod. V řádku 4 je deklarovaná počáteční hodnota srážky. Všimněte si, že jako příjemce zprávy je uvedena třída, ne objekt. Stejně tak v řádcích 12 a 13 je použito volání třídních metod. V řádku 17 je změněna hodnota srážky třídní metodou. Jak je vidět z příkladu, význam třídních metod a třídních atributů je především v tom, že třídní metody využíváme, aniž máme k dispozici objekt dané třídy a dále jsou hodnoty třídních atributů stejné pro všechny instance dané třídy.

Pro tisk bodů jsme využili formátovaný příkaz pro tisk **printf**. Tento příkaz vyžaduje, aby se tisk skládal ze dvou částí. **První část** tvoří formát a je uzavřena v uvozovkách a **druhou část** pak tvoří argumenty tisku. Uvozovací symbol pro formáty je znak ‚%', za kterým je uveden počet míst pro danou položku a následuje symbol typu tištěné proměnné **s** – pro řetězec a **d** – pro desítkové číslo **f** - pro reálné číslo. Formát %.2f znamená, že čísla před desetinnou čárkou nejsou nijak omezena, ale za desetinnou čárkou jsou vždy pouze dvě místa. Následuje další příklad formátu pro řetězce a celá čísla. Zároveň se vytisknou i všechny znaky, které jsou uvedeny v první části příkazy (v našem případě se jedná pouze o mezery).

```
String t = String.format("\n%11s %4s %4s %4d %4s %4d",
                  "Nazev
bodu:",getJmeno(),"X:",getX(),"Y:",getY());
```

Význam zápisu:

- tisk na nový řádek // \n
- 11 znaků pro text "Nazev bodu:" zarovnáno od leva
- 1 mezeta // mezera mezi %11s %4d
- 4 znaky pro jméno bodu
- 1 znak mezera

- 4 znaky pro text „X:“
- 1 znak mezera
- 4 znaky pro hodnotu x – souřadnici
- 1 znak mezera
- 4 znaky pro text „Y:“
- 1 znak mezera
- 4 znaky pro hodnotu y – souřadnici

Vyzkoušejte, co by se vytisklo, pokud bychom upravili uvedenou metodu následovně:

```
String t = String.format("\n%11s@%4s+%4sq%4dkk%4s*&$%4d",
                "Nazev
bodu:",getJmeno(),"X:",getX(),"Y:",getY());
```

Statické metody (třídní metody)

Doposud probírané metody byly tzv. instanční metody, to jsou metody, které se aplikují na objekty. Třídní metody (statické metody) se aplikují na danou třídu, zajišťují (nastavují) hodnoty třídních atributů, nebo se používají všude tam, kde nechceme uvádět příjemce jako objekt, ale místo příjemce uvádíme patřičnou třídu a k ní žádanou třídní metodu. Další charakteristiky jsou následující:

- definují se vložením klíčového slova **static** mezi identifikátory,
- mají platnost pro všechny instance (objekty) dané třídy,
- mohou být volány před vznikem instance dané třídy,
- slouží k přípravě prostředí, ve kterém vznikne objekt,
- slouží k definici metod, které nejsou vázány na žádnou instanci (objekt) - takto jsou definovány matematické funkce.

Třídní metody (Class methods) nezávisí na žádné instanci. V metodách třídy tedy **nemůžeme**:

- používat metody a atributy instancí – přesněji atributy a metody instancí kvalifikované klíčovým slovem this,
- překladač totiž nemá žádnou informaci, která instance by se za klíčovým slovem this mohla v daném okamžiku skrývat.

Výhodou těchto třídních metod je to, že při jejich aplikaci zadáme pouze název třídy, odpovídající třídní zprávu následovanou eventuálními argumenty. Praktické využití je například v použití knihovny *Math*, kdy většina metod této knihovny je deklarovaná jako třídní.
Aby uživatel nemohl vytvořit instance od dané třídy (není to v žádném případě žádoucí), je bezparametrický konstruktor uveden jako soukromý (private).

3.4 Metoda main

Za povšimnutí jistě stojí, že každá hlavní metoda main (metoda, která provádí požadované akce v aplikaci) je také deklarovaná jako třídní metoda (static).
Deklarací metody main jako static umožňuje JVM (Java Virtual Machine) vyvolat metodu main bez vytváření instance této třídy. Hlavička metody main je následující:

```
public static void main(String[] args)
```

Pro spuštění aplikace v dosovském okně zavoláme interpret Javy následovaný názvem třídy, tedy souboru, kde je uložena hlavní metoda a soubor je s příponou .class.

```
java JmenoClassName arg1 arg2 …// běh aplikace
```

JVM nahrává specifikovanou třídu (ClassName) a vyvolává metodu main této třídy. Proto je metoda main třídní.

3.5 Grafické prostředí pro jednoduché kreslení

Grafické prostředí je velmi vděčné prostředí pro skutečné pochopení objektově orientovaného přístupu. Navíc v úvodním kurzu do grafiky se setkáte nejen s grafickým prostředím, ale i s objektově orientovaným přístupem. Nemusíte se bát, v žádném případě nepůjde o překrývání obou kurzů. V našem kurzu se naučíte zobrazovat, mazat, posouvat *bod* a vykreslovat *čáru* a to vodorovně, svisle a pod úhlem 45 stupňů. Zbytek se dovíte v kurzu z grafiky.
Abychom si mohli zobrazit skutečně pouze jeden bod, potřebujeme vytvořit grafické prostředí. Java bohužel začíná se zobrazováním čáry, kdy je třeba zadat počáteční a koncový bod. Proto pro vykreslování si stáhnete z webu: www1.osu.cz/hunka záložka XOOP1 dva soubory a to: *BaseCanvas.java* a *Gui.java* do Vašeho projektu. Jedná se o dvě javovské třídy.
K vykreslování bodu budeme využívat třídu *Bod* z předchozí kapitoly, ale budeme ji muset rozšířit. Nejdříve se zaměříme na vysvětlení rozšíření datových atributů. **První část** kodu je následující:

Třída Bod – první část

```
import java.awt.Color;
1 public class Bod {
2   private int x;
3   private int y;
4   private Color barva;
5   private BaseCanvas obraz = Gui.getInstance().getCanvas();
6   private Color barvaPozadi = Color.white;
7   public Bod() {
8       this(0, 0, Color.red);
9    }
```

```
10   public Bod(int x, int y) {
11       this(x, y, Color.red);
12    }

13   public Bod(int x, int y, Color barva) {
14       this.x = x;
15       this.y = y;
16       this.barva = barva;
17    }

18   public Bod(int x, int y, String barva) {
19       this.x = x; this.y = y;
20       this.barva = this.stringToColor(barva);
21    }

22   // kopirovaci konstruktor
23   public Bod(Bod bod) {
24       x = bod.getX();
25       y = bod.getY();
26       barva = bod.getBarva();
27    } ...
```

Datový atribut na řádku 4 představuje barvu bodu. Využíváme k tomu knihovní třídu *Color*, kterou ale take musíme "importovat". Tato třída obsahuje základní barvy, jméno barvy je v angličtině. Barvy jsou ve třídě uloženy ve formě třídních (statických) datových atributů. Pokud tedy chceme zadat barvu bílou, zadáme `Color.white` kde *Color* je název třídy a *white* je datový atribut konkrétní barvy. Je to neobjektový přístup.
Jednou z možností, jak zadávat barvu je vytvořit se speciální třídu, která bude mít třídní (statické) metody, které převedou naše zadání barvy do podoby, které bude rozumět třída Color. Chceme tento převodník:
Barva.cervena() ⟶ Color.red
Barva.zluta() ⟶ Color.yellow

Tuto činnost provádí třída barva:

Třída Barva

```
import java.awt.Color;
public class Barva {
    public static Color cerna() {
        return Color.black;
    }
    public static Color cervena() {
        return Color.red;
    }
    public static Color modra() {
        return Color.blue;
    }
    public static Color zelena() {
        return Color.green;
    }
    public static Color zluta() {
```

```
            return Color.yellow;
        }
        public static Color ruzova() {
            return Color.pink;
        }
        public static Color oranzova() {
            return Color.orange;
        }
        public static Color seda() {
            return Color.gray;
        }
        public static Color modrozelena() {
            return Color.cyan;
        }
        public static Color bila() {
            return Color.white;
        }
        public static Color tmavoseda() {
            return Color.darkGray;
        }
        public static Color cervenoruda() {
            return Color.magenta;
        }
}
```

Uvedli jsme celou třídu, takže máte přehled o všech barvách. Jiná možnost jak zadat barvu je zadat ji jako řetězec, který v příkazu case vybere odpovídající barvu pro třídu *Color*. To řeší konstruktor, který je na řádku 18-21 třídy Bod. Konstruktor přímo volá metodu *stringToColor()*, která zabezpečí převod, viz výpis druhé části třídy *Bod*. Tím jsme vyřešili jeden datový atribut – barvu. Další datový atribut, který potřebujeme doplnit, je proměnná na kterou budeme kreslit. Ta se odkazuje na tzv. plátno – canvas. V řádku 5 třídy *Bod* deklarujeme proměnnou *obraz*, která splňuje tyto požadavky. Strukturu řádku necháme vždy beze změny, ale jméno proměnné můžeme samozřejmě deklarovat jiné.

Proměnná *obraz* je instance třídy *BaseCanvas*. Dále v řádku 6, třídy Bod musíme ještě deklarovat *barvu pozadí*, která bude bílá. Možná se Vám to zdá nadbytečné, ale v případě používání počítačů např. Apple, je toto třeba explicitně deklarovat, aby pozadí bylo všude skutečně bílé. Ve třídě *Bod* deklarováno celkem 5 konstruktorů.

- první konstruktor je bez parametrů; implicitně nastaví souřadnice na 0 a barvu na červenou;
- druhý konstruktor má jako parametry souřadnice nové instance; barva zůstává červená (tu jsme si sami zvolili pro dobrou viditelnost);
- třetí konstruktor má jako parametry souřadnice a parametr barva typu *Color*; vše se přiřadí přímo;

- čtvrtý konstruktor je stejný jako třetí konstruktor s tím rozdílem, že parametr barva je zadán jako řetězec; hodnoty řetězců musí odpovídat hodnotám v metodě *stringToColor()*, která je na řádcích 79-88; implicitní barva je červená.

Druhá část kódu třídy Bod:

```
28    public Color getBarva() {
29       return barva;
30    }

31   public BaseCanvas getObraz() {
32       return obraz;
33    }

34    public Color getBarvaPozadi() {
35        return barvaPozadi;
36    }

37   public void setBarva(Color barva) {
38       this.barva = barva;
39    }

40   public void vykresli() {
41       getObraz().putPixel(getX(), getY(), getBarva());
42    }
43   public void smaz() {
44       getObraz().putPixel(getX(), getY(),
                                    getBarvaPozadi());
45    }
46
47    public void vykresli(int x1, int y1) {
48        getObraz().putPixel(getX() + x1, getY() + y1,
                                          getBarva());
49    }

50    public void smaz(int x1, int y1) {
51        getObraz().putPixel(getX() + x1, getY() + y1,
                                    getBarvaPozadi());
52    }
53
54    public void posunSouradnic(int dx, int dy) {
55        setX(getX() + dx);
56        setY(getY() + dy);
57    }

58    public void posunBod(int dx, int dy) {
59        this.smaz();
60        this.posunSouradnic(dx, dy);
61        this.vykresli();
62    }
63
64    public String toString(){
65        return "X: " + getX() + " Y: " + getY() +
66         " barva: " + getBarva();
67    }
```

```
68    public void tisk() {
69        System.out.println(toString());
70    }

71    public void pauza(int doba) {
72        try {
73            Thread vlakno = new Thread();
74            vlakno.sleep(doba);
75        } catch(InterruptedException e) {
76            System.out.println("Chyba vlakno pauza");
77        }
78    }

79   public Color stringToColor(String barva) {
80       switch(barva) {
81           case "cerna": return Color.black;
82           case "cervena": return Color.red;
83           case "modra": return Color.blue;
84           case "zelena": return Color.green;
85           case "ruzova": return Color.pink;
86           default: return Color.red;
87       }
88    }

89   public int delkaX(Bod bod) {
90       return Math.abs(getX() - bod.getX());
91   }

92   public int delkaY(Bod bod) {
93       return Math.abs(getY() - bod.getY());
94    }
95 }
```

Nyní si vysvětlíme nejdůležitější metody **druhé části** třídy Bod.

Přístupová metoda *getObraz()* vrací odkaz na proměnnou obraz – tedy plátno, kde budeme kreslit.
Bezparametrická metoda *vykresli()* zobrazí daný bod na plátně (na které se odkazuje proměnná obraz). Uvnitř metody je metoda *putpixel()* jejíž parametry tvoří souřadnice bodu a barva (musí se explicitně uvést). Metoda *vykresli()* se dvěma parametry (x, y) zobrazí *bod* do zadaných souřadnic *x,y* s *barvou*, jakou má daný bod. Tato metoda slouží pro vykreslení čáry.

Oběma metodám vykresli odpovídají metody *smaz()* – bez parametrů, se dvěma parametry. Tyto metody fungují stejně jako metody vykresli s tím rozdílem, že barva pro vykreslení se rovná barvě pozadí. *Bod* vymažeme tím, že ho překreslíme barvou pozadí.

Metoda *pauza()* je pomocná metoda, která zbrzdí chod programu za každým vykresleným bodem malou pauzu a tak nám umožňuje vidět, jak vykreslování skutečně probíhá. Délka pauzy je přímo úměrná velikosti zadávané číselné konstanty do metody.

Obr. 3.4 Orientace os pro kreslení

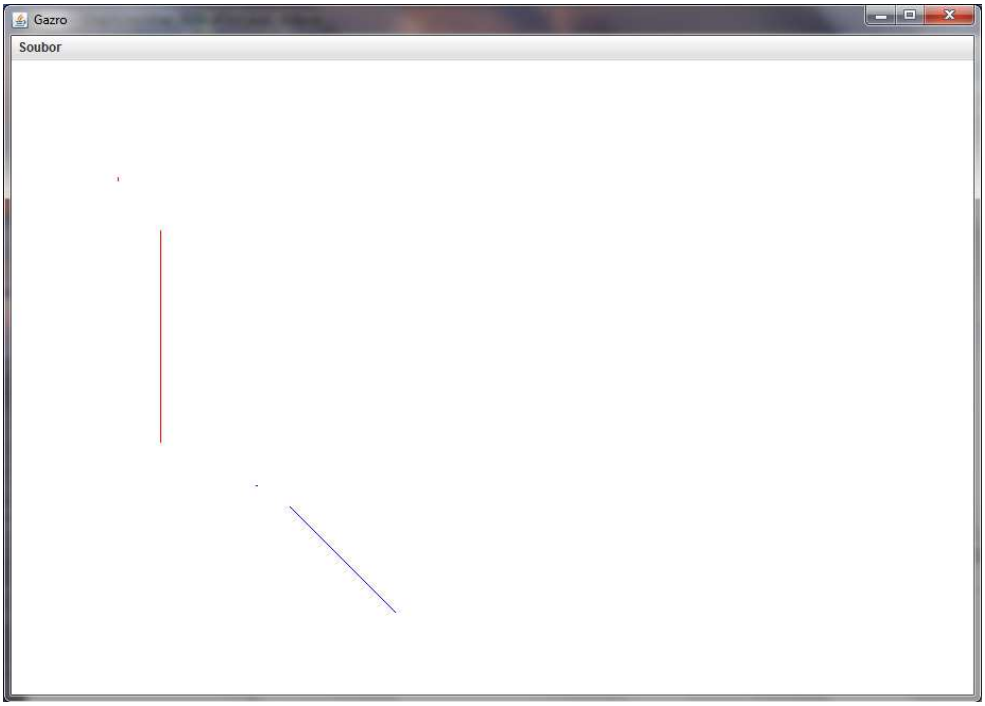
Než se dáme do kreslení, je třeba si uvědomit, že orientace os je trochu jiná, než jsme zvyklí z matematiky. Souřadnice (0, 0) je v levem hodním rohu.

Třída BodTest:

```
import java.awt.Color;
public class BodTest {
   public static void main(String[] args) {
       Bod bodA = new Bod(100,110, Color.red);
       Bod bodB = new Bod(230, 400, "modra");
       // vykresleni 4x bodA
       bodA.vykresli();
       bodA.vykresli(0, 1);
       bodA.vykresli(0, 2);
       bodA.vykresli(0, 3);
       bodA.pauza(20);

       // vykresleni 3x bodB
       bodB.vykresli();
       bodB.posunSouradnic(1, 0);
       bodB.vykresli();
       bodB.posunSouradnic(1, 0);
       bodB.vykresli();
       bodB.pauza(20);
       bodA.posunSouradnic(40, 50);

       // vykresleni svisle cary
       for(int i = 0; i < 200; i++) {
         bodA.vykresli(0, i);
         bodA.pauza(40);
       }
       bodB.posunBod(30, 20);

       // vykresleni vodorovne cary
       for(int i = 0; i < 100; i++) {
        bodB.vykresli(i, i);
        bodB.pauza(40);
       }
```

```
        // smazani cary
        for(int i = 0; i < 100; i++) {
         bodB.smaz(i, i);
         bodB.pauza(40);
      }
   }
}
```

Výsledek programu je na následujícím obrázku.



Obr. 3.5 Výsledek kreslení bodů a čar.

Protože je vykreslení jednoho bodu těžko postřehnutelné, zobrazujeme 4 body svisle (*bodA*) a 3 body vodorovně (*bodB*). Smazání modré čáry proběhne až na konci. Navíc při aplikaci vidíte názorně vykreslování jednotlivých bodů díky metodě pauza().

Tato kapitola se zabývala dalšími možnostmi, který nabízí objektově orientovaný přístup v Javě. Jedná se o pseudoproměnnou *this*, klíčové slovo *final*, třídní atributy a třídní proměnné. Dále kapitola uvádí třídu Bod rozšířenou tak, aby bylo možné instanci třídy Bod zobrazit.

Proč je deklarovaná metoda main jako statická?

V čem jsou výhody použití klíčového slova final pro datové atributy?

3.1 V čem jsou výhody třídních atributů?
3.2 V čem jsou výhody třídních metod?

3.1 Třídní atributy jsou přístupné všem objektům dané třídy buď přímo, nebo prostřednictvím třídních přístupových metod. Do těchto atributů je možné ukládat hodnoty, které vyžadují všechny objekty dané třídy. Např. třída *Teplota* může mít tři různé pohledy na teplotu a to prostřednictvím stupňů Celsia, Kelvina a Fahrenheita. Příslušná stupnice může být právě uložena v třídním atributu.

3.2 Výhodou třídních metod je to, že mohou být volány před vytvořením instancí dané třídy (jejich příjemcem je daná třída). Další výhodou je to, že se dají aplikovat na všechny instance dané třídy. Viz např. třída *Math* a její třídní metody. Vytváří zástupnou obálku ve které se mohou také deklarovat obecně potřebné metody.

4 Skládání objektů, grafické objekty (křížek, obdélník)

V této kapitole se dozvíte:

- jaké jsou základní možnosti skládání objektů,
- k čemu se využívá agregace,
- k čemu se využívá kompozice,
- jak vytvoříme složený objekt a jak se odkazujeme na jeho komponenty.

Po jejím prostudování budete schopni:

- rozumět významu pojmů agregace a kompozice,
- deklarovat složitější objekt složený z dílčích objektů,
- kreslit diagramy tříd pro skládání objektů.

Klíčová slova této kapitoly:

celek, část, agregace objektů, kompozice objektů

Klasifikace a kompozice

Klasifikace a kompozice (skládání) patří mezi základní prostředky, které nám pomáhají se vyrovnat se složitostí reálného světa. Protože naší snahou je vytvářet modely (modelovat) reálného světa v počítači (vytváříme tzv. referenční systém – viz kapitola 1), využíváme i zde prostředky klasifikace a kompozice.

Klasifikace představuje prostředek, pomocí kterého vytváříme a rozlišujeme mezi různými třídami jevů. To znamená, že vytváříme koncepty. Klasifikace umožňuje vytvářet hierarchie tříd a také umožňuje dědičnost mezi třídami. Detailněji se uvedené problematice budeme věnovat v kapitole o dědičnosti.

Kompozice – skládání je prostředek, který umožňuje, že jev může být chápán jako složenina (kompozice) dalších jevů. Je zde jasné rozlišení mezi jevem, který reprezentuje celek a jevy, reprezentující jeho části.

Jak jsme se již zmínili, klasifikace a kompozice jsou prostředky, pomocí kterých organizujeme složitost v pojmech hierarchií. Oba prostředky (klasifikace a kompozice) jsou používány v odlišným, ale navzájem se doplňujícím způsobem.

V této kapitole se budeme věnovat pouze skládání objektů, protože je relativně snadnější, dále pak v objektově orientovaném přístupu by skládání mělo mít přednost před dědičností. Dědičnost tříd (ke které se

ještě dostaneme je krásný a silný prostředek, ale není všespasitelný. Použije-li se místo dědičnosti ve třídě atribut, který bude podle potřeby odkazovat na instance různých tříd, dosáhne se tím v řadě případů větší flexibility, než při použití dědičnosti.

Existují dva základní typy skládání a to **agregace** a **kompozice**.

Agregace (někdy označovaná jako referenční kompozice) je volnou vazbou mezi objekty – vyskytuje se např. mezi počítačem a jeho periferními zařízeními, rezervace a odkazy na hotel, pokoj a hosta.

Kompozice je typem velmi těsné vazby mezi objekty – vyskytuje se např. mezi stromem a jeho listy, mezi autem a jeho jednotlivými součástmi (kola, karoserie, volant, motor ...), dřevěný panáček (hlava, krk, ruce, trup, nohy).

Pro skládání se používají následující termíny:

- celek nebo složený objekt
- části (složeného objektu).

Pro implementace těchto dvou základních typů skládání se v programovacích jazycích využívají dynamické resp. statické proměnné. Pro agregaci dynamické, referenční proměnné – umožňují vytvořit právě požadovanou volnou vazbu mezi celkem a částmi.

Pro kompozici se využívají statické proměnné, které jsou charakteristické svou „neměnností“. V jazyce Java (podobně jako v jazyce Smalltalk) však existují pouze dynamické proměnné. To do jisté míry zjednodušuje situaci, avšak neumožňuje explicitně využívat pro kompozici statické proměnné. Nebude tedy rozdíl mezi kompozicí a agregací tak zřetelný jako v jiných programovacích jazycích. Rozlišení mezi kompozicí a agregací probíhá na úrovni konstruktorů. Pro kompozici platí, že vazba mezi celkem a jeho částmi se vytváří v konstruktoru. Naopak pro agregaci platí, že vazba mezi celkem a jeho částmi se vytváří následně kdekoli v programu, resp. může být pak následně kdekoli v programu změněna, což umožňuje flexibilita.

Obecná struktura třídy pro skládání

```
public class Celek {

	private CastA castA;
	private CastB castB;
	private CastC castC;

	public Celek() {
	 // konstruktor;
	}
```

```
        public kvalifikátor metoda1() {
            castA.metodaA_1();
            //  . . .
            castB.metodaB_4();
            }

        public kvalifikátor metoda2() {
                  // . . .
            castC.metodaC_7();
        }
}
```

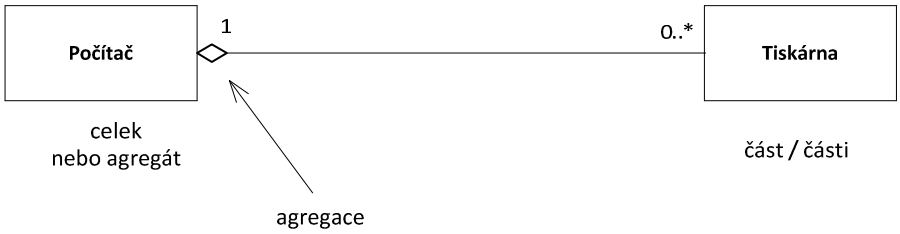
Uvedený výpis obsahuje deklaraci třídy Celek. Části tohoto celku jsou deklarovány pomocí datových atributů patřičných tříd. Např. datový atribut castA je instancí třídy CastA atd. Celek využívá své části ve svých metodách. Např. uvnitř metody metoda1 jsou vyvolány metody metodaA_1 pro objekt castA a metodaB_4 pro objekt castB.

Kvalifikátor v popisu metod představuje patřičný návratový typ příslušné metody, nebo void pro metodu která nic nevrací.

4.1 Sémantika agregace

Agregace je relací typu celek / část. Část reprezentuje jednu až mnoho částí. V tomto typu relace používá jeden objekt (celek) služby dalšího objektu (části).

- Celek:
  - bývá dominantní v relaci,
  - řídí chod relace.
- Části:
  - poskytují služby,
  - reagují na požadavky celku (jsou pasivní).



Obr. 4.1 Znázornění agregace pomocí diagramu tříd UML

Na obr. 4.1 je znázorněna agregace mezi objektem Počítač (reprezentuje celek) a objektem Tiskárna (reprezentuje část. Celek má u sebe značku agregace, což je prázdný kosočtverec. Z diagramu je také vidět, že celek může být jeden, nebo žádný a části mohou být žádné až nekonečně mnoho (znázorněno symbolem hvězdičky).

Slovní vyjádření obrázku 4.1:

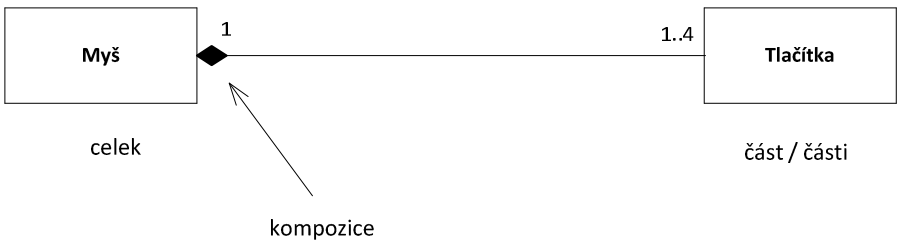
- k Počítači může být připojeno více tiskáren, ale i žádná
- Tiskárna může být připojena max. k jednomu počítači, nebo žádnému počítači
- danou Tiskárnu může postupně používat více Počítačů
- Tiskárna je v podstatě na Počítači nezávislá

Shrnutí agregace

- Celek (agregát) bývá závislý na částech, ale může existovat nezávisle na nich (občas také existuje)
- Části mohou existovat nezávisle na celku
- Chybí-li některé části, je celek v jistém smyslu neúplný.
- Části mohou být sdíleny více celky.

4.2 Sémantika kompozice

- Kompozice je silnější formou relace celek / část.
- V kompozici nemohou části existovat mimo celek.
- V kompozici každá část patří pouze jednomu celku.



Obr. 4.2 Znázornění kompozice pomocí diagramu tříd UML

Z obrázku 4.2 je patrné, že objekt Myš je složen z jednoho až čtyř tlačítek, které k objektu Myš patří a nemohou bez něho existovat. Celek má u sebe značku, kterou v případě kompozice tvoří plný kosočtverec.

Shrnutí kompozice

- Části patří výhradně jen jednomu celku (kompozici).
- Celek nese výhradní odpovědnost za použití všech svých částí – znamená to zodpovědnost za jejich tvorbu a zničení.
- Za předpokladu, že odpovědnost za části přejde na jiný objekt, může celek tyto části uvolnit.
- Je-li celek zničen, musí zničit rovněž všechny svoje součásti, nebo převést odpovědnost za ně na nějaký další objekt.
- Vzhledem k tomu, že v kompozici má celek výhradní odpovědnost nejen za životní cyklus, ale i za uspořádání všech svých částí, vytvoří složený objekt při svém vytvoření všechny své části automaticky.
- Podobně je tomu i při vymazání objektu. Složený objekt (celek) musí ještě před svým vymazáním vymazat (zničit) všechny své části, nebo je uspořádat tak, aby je mohl přijmout jiný složený objekt (celek).

4.3 Objekty křížek a obdélník z pohledu skládání

Nyní uvedeme pár konkrétních příkladů.
Následující příklad zobrazuje jednoduchý křížek. Třída Krizek má následující deklaraci:

```
public class Krizek {
   private Bod stred;
   private int delka = 3;

   public Krizek() {
       stred = new Bod();
    }

   public Krizek(Bod bod, int delka) {
       //stred = bod; - normalni prirazeni

       // pouziti kopirivaciho konstruktoru tridy Bod
       stred = new Bod(bod);
       this.delka = delka;
    }

   // kopirovaci konstruktor
   public Krizek(Krizek kr) {
       stred = new Bod(kr.getStred());
       delka = kr.getDelka();
    }

   //modifikacni a pristupove metody
   public Bod getStred() {
       return stred;
    }
```

```
    public int getDelka() {
        return delka;
     }

    public void setStred(Bod bod) {
        stred = bod;
     }

    public void setDelka(int delka) {
        this.delka = delka;
     }

    public void vykresli() {
        for(int i = 0; i < getDelka() * 2 + 1; i++){
            //vodorovne
            getStred().vykresli( - getDelka() + i, 0);
            //svisle
            getStred().vykresli(0, - getDelka() + i);
            getStred().pauza(20);
         }
     }

    public void smaz() {
        for(int i = 0; i < getDelka() * 2 + 1; i++){
            //vodorovne
            getStred().smaz( - getDelka() + i, 0);
            //svisle
            getStred().smaz(0, - getDelka() + i);
            getStred().pauza(20);
         }
     }

    public void posun(int dx, int dy) {
        this.smaz();
        this.getStred().posunSouradnic(dx, dy);
        this.vykresli();
     }

    public void posunSouradnic(int dx, int dy) {
        this.getStred().posunSouradnic(dx, dy);
     }

    public void posunNemas(int dx, int dy) {
        this.getStred().posunSouradnic(dx, dy);
        this.vykresli();
     }

    public String toString() {
        return "Stred: " + getStred().toString() + " delka: " +
                           getDelka();
        }

    public void tisk() {
        System.out.println(toString());
     }
}
```

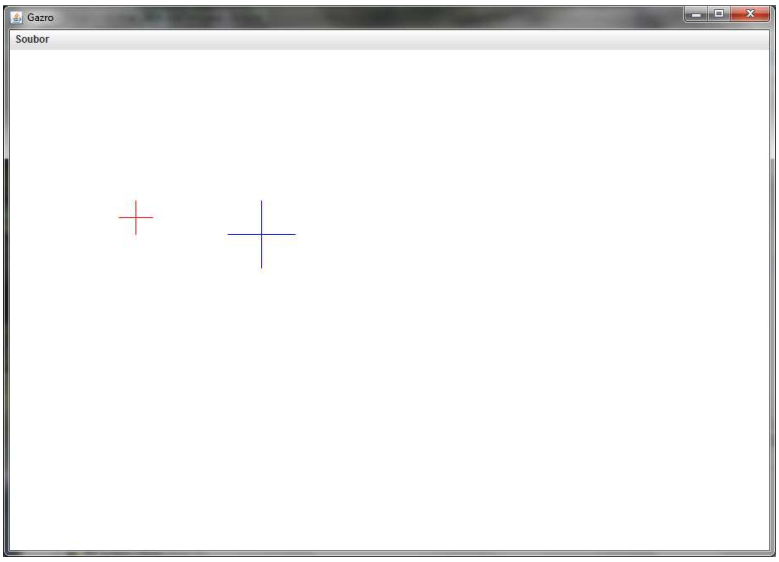
Třída *Krizek* má dva datové atributy. Prvním atributem je *stred*, což je proměnná třídy *Bod* označující střed křížku. Druhým atributem je *délka* strany křížku. Jedná se o kompozici. Celkem je instance třídy Krizek a část je instance třídy Bod.

Obr. 4.3 Kompozice celek část

Využití je ukázáno ve třídě KrizekTest, která kreslí a maže křížky.

Třída KrizekTest

```
public class KrizekTest {
    public static void main(String[] args) {
        Bod bod = new Bod(100, 120, Color.blue);
        Krizek krizek = new Krizek(bod, 40);
        krizek.vykresli();
        bod.pauza(1500);
        krizek.posun(200, 100);

        Krizek krizekA =
              new Krizek(new Bod(150, 200, "cervena"), 20);
        krizekA.vykresli();
    }
}
```



Obr. 4.4 Výstup třídy *KrizekTest*

Následující příklad je ukázkou kompozice, kdy každý objekt třídy Obdélník je složen ze dvou objektů třídy Bod a sice z *levyHorni* reprezentující levý horní bod obdélníka a *pravyDolni* reprezentující pravý dolní bod obdélníka. Pro úplnost ještě uvedeme diagram tříd jazyka UML.

Obr. 4.5 Kompozice – příklad

Třída Obdelnik

```
public class Obdelnik {
   private Bod levyHorni;
   private Bod pravyDolni;

   //konstruktory
   public Obdelnik() {
       levyHorni = new Bod();
       pravyDolni = new Bod();
    }

   public Obdelnik(Bod a, Bod b) {
       //levyHorni = a;
       //pravyDolni = b;
       levyHorni = new Bod(a);
       pravyDolni = new Bod(b);
    }

   public Obdelnik(Obdelnik ob) {
       levyHorni = new Bod(ob.getLevyHorni());
       pravyDolni = new Bod(ob.getPravyDolni());
    }

   // pristupove metody
   public Bod getLevyHorni() {
       return levyHorni;
    }

   public Bod getPravyDolni() {
       return pravyDolni;
    }

   public void setLevyHorni(Bod bod) {
       levyHorni = bod;
       //levyHorni = new Bod(bod);
    }

    public void setPravyDolni(Bod bod) {
        pravyDolni = bod;
    }

   public String toString() {
       return "Obdelnik levyHorni: " +
              getLevyHorni().toString() +
                " pravyDolni: " + getPravyDolni().toString();
   }
```

```
    public void tisk() {
         System.out.println(toString());
     }

    public int getDelkaX() {
        return Math.abs(getLevyHorni().getX() –
                            getPravyDolni().getX());
     }

     public int getDelkaY() {
         return Math.abs(getLevyHorni().getY() –
                              getPravyDolni().getY());
     }

    public void vykresli() {
        // kreslí vodorovné strany obdelníka
        for(int i = 0; i <= getDelkaX(); i++) {
            getLevyHorni().vykresli(i, 0);
            getPravyDolni().vykresli(-i, 0);
            getPravyDolni().pauza(20);
         }

        // kreslí svislé strany obdelníka
        for(int i = 0; i <= getDelkaY(); i++) {
            getLevyHorni().vykresli(0, i);
            getPravyDolni().vykresli(0, -i);
            getLevyHorni().pauza(20);
         }
     }

    public void smaz() {
        // maže vodorovné strany obdelníka
        for(int i = 0; i <= getDelkaX(); i++) {
            getLevyHorni().smaz(i, 0);
            getPravyDolni().smaz(-i, 0);
            getPravyDolni().pauza(20);
         }

        // maže svislé strany obdelníka
        for(int i = 0; i <= getDelkaY(); i++) {
            getLevyHorni().smaz(0, i);
            getPravyDolni().smaz(0, -i);
            getLevyHorni().pauza(20);
         }
     }

    public void posun(int dx, int dy) {
        this.smaz();
        this.posunSouradnic(dx, dy);
        this.vykresli();
     }

    public void posunSouradnic(int dx, int dy) {
        getLevyHorni().posunSouradnic(dx, dy);
        getPravyDolni().posunSouradnic(dx, dy);
     }
}
```

Metoda *vykresli()* je složena ze dvou částí. První cyklus for kreslí vodorovné stany obdélníka a druhý cyklus for kreslí svislé strany obdélníka. Dále také vidíme, že při důsledném objektově orientovaném postupu je metoda *posun* docela jednoduchá. Skládá se z metod *smaz(), posunSouradnic()* a *vykresli().*

Platí zásada, že objekt je samostatná sebe identifikovatelná entita a podle toho bychom také měli postupovat a svěřit vždy příslušnou „odpovědnost“ patřičnému objektu. Stane-li se, že objekt typu část námi požadované metody nemá, tak tyto metody doplníme.

Obdobně jako v metodě posun je postupováno i v metodě *toString()*, kde při vykonání celé metody jsou dílčí úkoly delegovány na dva objekty třídy Bod. Názvy metod *posun* a *toString()* jsou záměrně stejné, protože se vždy jedná o jiné třídy (třídu *Obdélník* a třídu *Bod*).

Doplňte třídu *ObdelnikTest*, která zobrazí první obdélník s body (40, 60), (150, 170) v barvě červené a druhý obdélník s body (100, 120), (250, 300) v barvě modré. Proveďte posun obdélníků o (30, 50) resp. (-30, - 40).

V této kapitole jste se seznámili, jaké jsou možnosti skládání objektů do celků. Prakticky jste viděli, jak se vytvoří složený objekt křížek a složený objekt obdélník.

Jaká zásada platí, když celek požaduje nějakou operaci po části?

Celek v žádném případě nevykonává požadovanou operaci přímo, ale nejbližšímu objektu, který představuje část, zašle zprávu požadující operaci. Tato část, která pro eventuální nižší část představuje celek eventuálně opět zašle zprávu objektu části, ze kterého se skládá celek. Tím se sníží chybovost a zaručí se přehlednost.

Jaké operace jsou třeba provést, když posouváme objekt na jiné místo?

5 Skládání objektů (účet, osoba, adresa), přetížené konstruktory

V této kapitole se dozvíte:

- jak vytvářet v Javě kompozici a jak vytvářet agregaci,
- jak vyvolávat z jednoho objektu metody druhého objektu,
- jak deklarujeme přetížené konstruktory ve třídách.

Po jejím prostudování budete schopni:

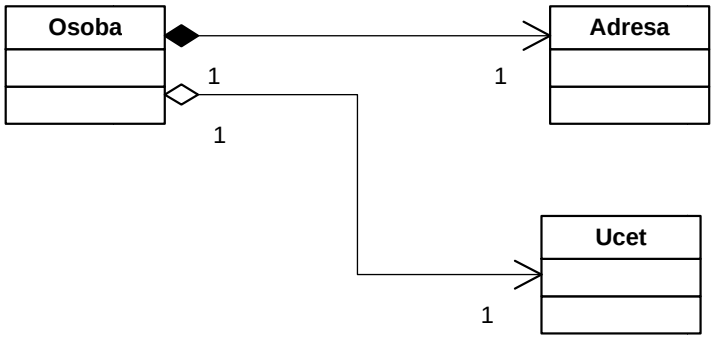
- lépe rozumět pojmům objekt, třída, budete vědět, jak se třída graficky zobrazuje v diagramu tříd UML, vytvořit a spustit jednoduchý objektově orientovaný program.

Klíčová slova této kapitoly:

přetížené konstruktory, pseudoproměnná this

5.1 Skládání objektů (osoba, adresa, účet)

Další trochu složitější příklad bude ukázka kompozice i agregace. Na diagramu tříd UML je zobrazena celá problematika příkladu.



Obr. 5.1 Ukázka kompozice a agregace

Objekty třídy *Osoba* jsou složeny ze dvou částí, a sice z objektu třídy *Adresa* (skládání pomocí kompozice – pevná vazba daná již v konstruktoru metody) a objektu třídy *Ucet* (skládání pomocí agregace – volnější, pružnější vazba). Protože jazyk Java má všechny proměnné dynamické, využíváme ke skládání formou kompozice kopírovací konstruktor.

Nejdříve uvedeme deklarace obou tříd, jejichž objekty tvoří části a pak teprve třídu *Osoba* tvořící celek.

Třída Adresa

```
public class Adresa {
   private String ulice;
   private int cislo;
   private String mesto;

    // konstruktory deklarace
   public Adresa() {
     ulice = "nezadana"; cislo = 0; mesto = "neuvedene";
    }

   public Adresa(String ulice, int cislo, String mesto)  {
       this.ulice = ulice;
       this.cislo = cislo;
       this.mesto= mesto;
   }

   public Adresa(Adresa ad){ // kopirovaci konstruktor
      this.setUlice(ad.getUlice());
      this.setCislo(ad.getCislo());
      this.setMesto(ad.getMesto());
   }

   // pristupove metody . . .

   // modifikacni metody . . .

   public String toString(){
      return String.format("Ulice: %s cislo: %4d mesto: %s",
            getUlice(), getCislo(), getMesto());
   }

   public void tisk() {
      System.out.println( this.toString());
   }

   public void setAdresa(Adresa a) {
      this.setUlice(a.getUlice());
      this.setCislo(a.getCislo());
      this.setMesto(a.getMesto());
   }
}
```

Pro úsporu místa nejsou v deklaraci třídy uvedeny přístupové a modifikační metody (get, set).

Třída Ucet

```
public class Ucet {
   private int cislo;
   private int stav;

   // Konstruktory tridy Ucet
   public Ucet() {
      this(0, 0);
    }
```

```
    public Ucet(int cislo, int stav) {
        this.cislo = cislo;
         this.stav = stav;
        }

    public void vlozeni (int castka) {
        stav = stav + castka;
        }

    public int vyber (int castka) {
         stav = stav - castka;
         return stav;
      }

    // pristupove a modifikacni metody . . .

    public String toString() {
         return String.format("Cislo uctu: %d stav uctu: %d",
              getCislo(), getStav());
      }

    public void tisk() {
         System.out.println(this.toString());
         }
}
```

Nejdůležitějšími metodami jsou u třídy *Ucet* metody *vlozeni* a *vyber*. Bylo by samozřejmě potřebné také doplnit přístupové a modifikační metody.

```
public class Osoba {
    private String jmeno;
    private int rokNarozeni;
    private Adresa adresa;  //kompozice
    private Ucet ucet;      // agregace

    // deklarace konstruktoru
    public Osoba()  {
        this("neuvedeno", 1900, new Adresa(), new Ucet());
    }

    public Osoba(String jmeno, int rokNarozeni, Adresa adresa,
                                                Ucet ucet) {
        this.jmeno= jmeno; this.rokNarozeni = rokNarozeni;
       //kopirovaci konstruktor
       this.adresa = new Adresa(adresa);
       //this.adresa = adresa; odkaz je pristupny
       this.ucet = ucet; //agregace
    }

    // pristupove a modifikacni metody . . .
```

```
    public String toString() {
       String tx= String.format("\nJmeno: %s rok narozeni: “+
            “%4d\nAdresa: %s\nUcet: %s", getJmeno(),
             getRokNarozeni(), adresa.toString(),
             ucet.toString());
     return tx;
    }

    public void tisk() {
        System.out.println(this.toString());
      //adresa.tisk(); ucet.tisk();
      // jiz deklarovana v metode toString
    }

    public void setUcet(Ucet ucet) {
        this.ucet = ucet;
    }

    public void setAdresa(Adresa a) {
        adresa.setAdresa(a);
    }

    public Ucet getUcet(){
        return ucet;
    }

    public void vlozeni(int castka){
        ucet.vlozeni(castka);
    }

    public void vyber(int castka) {
        ucet.vyber(castka);
    }

    public void tiskUcet(){
        System.out.println("Ucet: "+ucet.toString());
    }

    public void tiskAdresa(){
        System.out.println("Adresa: "+ adresa.toString());
    }
}
```

U třídy osoba nejsou již uvedeny přístupové a modifikační metody (get, set) pro jednoduché datové atributy deklarované v této třídě. Všimněte si také, jak se třída *Osoba* ve svých metodách obrací k objektům tříd *Adresa* a *Ucet*.

A nyní ke kompozice a agregaci. Třída *Osoba* využívá kompozice pro skládání s objektem třídy *Adresa*. Vždy v obou deklarovaných konstruktorech je nutné zadat objekt třídy *Adresa*. Objekt třídy *Ucet* zadán není a proto zůstává jeho odkaz neobsazen, tedy ukazuje na null.

Objekt třídy *Ucet* je do třídy *Osoba* dodán pomocí modifikační metody *setUcet*. Tak je také možno vyměnit stávající objekt třídy *Ucet* za jiný

objekt třídy *Ucet*. Pro úplnost si ještě uvedeme výpis programu *OsobaTest*, který otestuje celou aplikaci.

Třída OsobaTest

```
public class OsobaTest {
   public static void main(String[] args) {
       Adresa a1 = new Adresa("Na nabrezi",237,"Havirov");
      Osoba o1 = new Osoba("Kamil",1988, a1, new Ucet());
      Ucet u1 = new Ucet(1,200);
      Osoba o2 = new Osoba("Petr",1956,
                  new Adresa("30. dubna",22,"Ostrava"), u1);
      Ucet u2 = new Ucet(2, 300);
      o1.setUcet(u2);
      o2.setUcet(u1);
      o1.tisk();
      o2.tisk();
      o1.vyber(300);
      o2.vlozeni(800);
   }
}
```

Jedná se jen o krátký test funkčnosti agregace a kompozice. Tento příklad je možné dále rozvíjet, přidávat další objekty.
Protože třída *Osoba* má deklarovaný buď bezparametrický konstruktor, anebo konstruktor se všemi parametry, v případě, že nemáme deklarovanou instanci třídy *Ucet*, doplníme *new Ucet()*, přímo do parametru konstruktoru.

5.2 Deklarace tříd s přetíženými konstruktory

Jak jste již sami viděli, můžete si deklarovat své vlastní konstruktory. Důležité je ukázat, jak se dají konstruktory přetěžovat. Pro přetěžování se využívá pseudoproměnná this, která v dané třídě znamená, že odkazuje sama na sebe. Takže pokud je v následujícím příkladu uvedeno v těle konstruktoru
this(0, 0, 0);

znamená to, že se vyvolá konstruktor dané třídy, který očekává tři parametry, do nichž se dosadí v daním případě samé nuly. Je to přehlednější a lépe udržovatelné. Následující příklad je toho praktickou ukázkou.

Kromě toho je také v příkladu vidět, jak může fungovat konstruktor, kterému předáváme celý objekt dané třídy.

Třída Time

```
public class Time {
   private int hodina;   // 0 - 23
   private int minuta; // 0 - 59
   private int vterina; // 0 – 59
```

```
 // konstruktory
 public Time() {
    this( 0, 0, 0 );
}

public Time( int h ) {
    this( h, 0, 0 );
}

public Time( int h, int m ) {
    this( h, m, 0 );
}

public Time( int h, int m, int v ) {
     // vyvola metodu setTime k validaci casu
    setTime( h, m, v );
}

// kopirovaci konstruktor
public Time( Time time ) {
   this( time.getHodina(), time.getMinuta(),
                           time.getVterina() );
}

public void setTime( int h, int m, int v ) {
    setHodina( h );
   setMinuta( m );
   setVterina( v);
    }

public void setHodina( int h ) {
    hodina = ( ( h >= 0 && h < 24 ) ? h : 0 );
    }

public void setMinuta( int m ) {
    minuta = ( ( m >= 0 && m < 60 ) ? m : 0 );
    }

public void setVterina( int v ) {
    vterina = ( ( v >= 0 && v < 60 ) ? v : 0 );
    }

public int getHodina() {
    return hodina;
    }

public int getMinuta() {
    return minuta;
    }

public int getVterina() {
    return vterina;
    }
```

```
   // prevede na String v universalnim casovem
   // formatu (HH:MM:SS)
   public String toUniversalString() {
      return String.format("%02d:%02d:%02d",
         getHodina(), getMinuta(), getVterina() );
   }

   // prevede na String ve standardnim casovem formatu
   // (H:MM:SS AM nebo PM)
   public String toString() {
      return String.format( "%d:%02d:%02d %s",
         ( (getHodina() == 0 || getHodina() == 12) ? 12 :
          getHodina() % 12 ),
         getMinuta(), getVterina(),
            ( getHodina() < 12 ? "AM" : "PM" ) );
   }

   public void printU() {
      System.out.println("   "+this.toUniversalString());
      }

   public void printS() {
      System.out.println("   "+this.toString());
      }

   public void tisk() {
      System.out.printf("%s %03d %2d %2d","Aktualni cas: ",
               getHodina(),getMinuta(), getVterina());
      }
}
```

Testovací program třídy *TimeTest*. V něm jsme použili metodu *printU*( ) a *printS*( ) pro zkrácený výpis výsledků. V komentářích je uveden celý odpovídající kód.

Třída TimeTest

```
public class TimeTest {
   public static void main(String[] args) {
      Time t1 = new Time();             // 00:00:00
      Time t2 = new Time( 2 );          // 02:00:00
      Time t3 = new Time( 21, 34 );     // 21:34:00
      Time t4 = new Time( 12, 25, 42 ); // 12:25:42
      Time t5 = new Time( 27, 74, 99 ); // 00:00:00
      Time t6 = new Time( t4 );         // 12:25:42

      System.out.println( "Vytvoreno s:" );
      System.out.println( "t1: vsechny argumenty defaultni" );
      t1.printU();
      t1.printS();
      System.out.println(
      "t2: specifikovana hodina; minuta a vterina defaultni");
       t2.printU();
       t2.printS();

       System.out.println(
       "t3: specifikovana hodina a minuta; vterina defaultni");
```

```
        t3.printU();
        t3.printS();

        System.out.println(
                "t4: specifikovana hodina, minuta a vterina");
        t4.printU();
        t4.printS();

        System.out.println(
            "t5: specifikovany vsechny neplatne hodnoty" );
        t5.printU();
        t5.printS();

        System.out.println(
            "t6: ma specifikovane hodnoty objekty t4" );
        t6.printU();
        t6.printS();
        }
}
```

V této kapitole jste viděli názorný příklad využití skládání objektů. Jednalo se o kompozici a agrageci. Protože jazyk Java má všechny proměnné dynamické (neobsahují objekty, ale na ně ukazují) spočívá řešení v tom, že pro kompozici je objekt přímo deklarovaný při deklaraci datového atributu, eventuálně je inicializován pomocí kopírovacího konstruktoru.
Pro agregaci použijeme přiřazení odkazů, tedy dvě proměnné odkazují na stejný objekt.

Při analýze je třeba jasně specifikovat, zda celek může existovat bez svých částí, či ne. Při skládání objektů je velmi důležité si uvědomit, co je celek a co jsou jeho části. Dále se musíme zaměřit na vazby mezi celkem a částmi a z toho pak dostaneme výsledek, zda se bude jednat o kompozici objektů nebo agregaci.

Jak se deklarují přetížené konstruktory a na co se používá pseudoproměnná this.

Kompozice a agregace hrají velmi důležitou úlohu při analýze a návrhu programových systémů. Protože Java nepoužívá „statické“ proměnné,

je mezi oběma formami skládání objektů menší rozdíl, než v jiných jazycích. Je dobré se zprvu zaměřit při analýze problému na skládání a teprve později na dědičnost. Skládání mnohdy přináší efektivnější a přirozenější model než pouhé využívání dědičnosti.

6 Práce s poli jako datovými atributy

V této kapitole se dozvíte:

- jak se dají využívat pole jako datové atributy tříd,
- že přístupové a modifikační metody jsou stejné jako pro jiné datové atributy,
- jak se dají využít výčtové typy pro aplikace s poli,
- jak se jednoduše pole převede na kolekci.

Po jejím prostudování budete schopni:

- lépe využívat pole jako datové atributy,
- rozhodnout, zda využít výčtové typy,
- lépe se orientovat v reálných aplikacích.

Klíčová slova této kapitoly:

datový atribut jako pole, výčtový typ, výčtové typy a pole

Práce s poli jako datovými atributy zůstává spíše opomíjená. Přitom s datovými atributy typu pole pracujeme stejným způsobem jako s jinými datovými atributy. Samozřejmě při práci s poli je třeba si uvědomit, že když deklarujeme pole, musíme nadeklarovat všechny jeho prvky. Tím je míněno, že nemůžeme obsadit v poli jen část a zbytek nechat neobsazen. Jak tento problém vyřešit ukazujeme v kapitole 10, kde se zabýváme tzv. registry. Registr je třída, která má jako datový atribut pole.

6.1 Případová studie – zkoušky, oblíbená jídla

Nejdříve si uvedeme příklad třídy *Student*

Třída Student

```
import javax.swing.JOptionPane;
public class Student {
   private String jmeno;
   private int[] zkousky = {1, 3, 2, 2, 1};
   private String[] oblibenaJidla;

   public Student() {
       jmeno = "nezadane";
       oblibenaJidla = new String[1];
       oblibenaJidla[0] = "nezadana";
    }
```

```
public Student(String jmeno){
     this.jmeno = jmeno;
     initOblibenaJidla();
}

public String getJmeno() {
    return jmeno;
}

public String[] getOblibenaJidla(){
     return oblibenaJidla;
}

public int[] getZkousky() {
   return zkousky;
}

public void setJmeno(String jmeno) {
    this.jmeno = jmeno;
}

public void setOblibenaJidla(String[] jidla) {
     oblibenaJidla = jidla;
}

public void setZkousky(int[] zkousky) {
     this.zkousky = zkousky;
}

public String toString() {
  return String.format("Jmeno: %s \nOblibena jidla “ +
   “%s\nZkousky %s\nPrumer: %.2f",
   getJmeno(), jidlaToString(), zkouskyToString(),
    getPrumer());
}

public String jidlaToString() {
    String s = "";
    for(int i = 0; i < getOblibenaJidla().length; i++)
       s += "\n" + (i + 1) + "  " + oblibenaJidla[i];
    return s;
}

public String zkouskyToString() {
   String s = "";
   for(int i = 0; i < getZkousky().length; i++)
      s += "\n" + (i + 1) + "  " + zkousky[i];
   return s;
}

public void tisk() {
    System.out.println(toString());
}
```

```
   public void initOblibenaJidla() {
     String odpoved =
      JOptionPane.showInputDialog("Pocet oblibenych jidel: ");

      int pocet = Integer.parseInt(odpoved);
      oblibenaJidla = new String[pocet];
      for(int i = 0; i < pocet; i++) {
        odpoved =
        JOptionPane.showInputDialog("Jidlo [" + (i+1) + "] ");
        oblibenaJidla[i] = odpoved;
        }
   }

   public double getPrumer() {
       double vysledek = 0;
       for (int i = 0; i < getZkousky().length; i++)
              vysledek += zkousky[i];
       vysledek = vysledek / getZkousky().length;
       return vysledek;
   }

   public String prumerToString() {
        return ""+ getPrumer();
   }
}
```

Jak je vidět z kódu třídy *Student*, jsou v ní deklarované dva datové atributy. První s identifikátorem *zkoušky*, obsahuje celočíselné hodnoty v rozsahu 1..3. Tyto hodnoty jsou deklarovány přímo při deklaraci datového atributu. Druhým datovým atributem je pole s identifikátorem *oblibennaJidla*, obsahující řetězcové hodnoty oblíbených jídel.

Zadávání rozsahu pole a jednotlivých prvků pole se provádí v metodě initOblibenaJidla(). Tato metoda využívá třídu *JOptionPane* a její třídní (statickou metodu – netřeba vytvářet instanci) *showInputDialog()*. Tato metoda vrací odpověď ve formě proměnné typu String. Konkrétní proměnná v uvedené metodě se jmenuje odpověď. Pokud očekáváme odpověď datového typu Integer. je třeba provést převod řetězce na celé číslo. To provádí metoda *parseInt* obalové třídy *Integer*. S využitím metody *initOblibenaJidla()* zadá uživatel nejdříve počet svých oblíbených jídel a pak konkrétní oblíbená jídla.

Metoda *getPrumer()* vrátí průměrnou hodnotu dosaženou ve zkouškách.

Třída StudentTest

```
1 public class StudentTest {
2   public static void main(String[] args) {
3      Student student = new Student("Eva");
4      student.tisk();
5      String[] jidlo = {"cocka s vejci", "vepro knedlo zelo",
6                                "michana zelenina"};
```

```
7       int[] vysledky = {3, 3, 2, 1 };
8       student.setOblibenaJidla(jidlo);
9       student.setZkousky(vysledky);
10       student.tisk();
        }
}
```

Třída *StudentTest* nejdříve provede tisk jména studenta, jeho zkoušek a oblíbených jídel. Na řádcích 5 a 6 deklarujeme pole jídlo s konkrétními hodnotami.
V řádku 7 deklarujeme pole výsledky s konkrétními celočíselnými hodnotami. Obě uvedená pole nahradí původní datové atributy ve třídě Student. Využijí se k tomu metody *setOblibenaJidla()* a *setZkousky()*.

6.2 Výčtové typy

Než se pustíme do dalšího příkladu, který simuluje 52 karet a jejich míchání, vysvětlíme si výčtové typy. Výčtový typ je v Javě speciální třída s označením enum. Používá se k zadávání hodnot, které se nemění. V našem případě se jedná o 4 různé barvy ve standardních kartách a o jednotlivé karty. Metoda *values()* vrací pole hodnot výčtového typu. Výčtový typ *Barva* bude tedy sloužit jak v úvodním názorném příkladu, jak se pracuje s výčtovými typy a dále bude také použit v aplikaci *BalicekKaret*.

```
public enum Barva {
      Krize, Herce, Kary, Piky
}
```

```
1 public class BarvaTest {
2   public static void main(String[] args) {
3     Barva bp = Barva.Piky;
4     Barva bh = Barva.Herce;
5     if(bp == Barva.Piky) System.out.println("ANO Piky");
6                   else System.out.println("Ne Piky");

7     // jedna moznost pruchodu vyctovymi hodnotami
8     for(Barva bv: Barva.values())
9           System.out.println("Barva: " + bv);

10    System.out.println(); // prazdny radek ve vypisu
11    // klasicka moznost pruchodu vyctovymi hodnotami
12    Barva[] pole = Barva.values();
13    for(int i = 0; i < pole.length; i++)
14          System.out.println("Barva: " + pole[i]);
    }
}
```

Ve třídě *BarvaTest* vidíme, že do proměnných typu *Barva* můžeme přiřadit konkrétní výčtový typ a pak uvedenou proměnnou např. testovat, zda obsahuje konkrétní výčtový typ. Jak bylo zmíněno, metoda values() vrací pole výčtových hodnot viz řádek 12. Pak můžeme procházet klasicky přes všechny hodnoty daného výčtového

typu. Řádek 9 i řádek 14 tisknou všechny hodnoty výčtového typu barva, tedy čtyři barvy karet.

Následující třídy spolu s výčtovým typem *Barva* ukazují, jak využijeme pole pro modelování 52 hracích karet. Víme, že karty mají jednak barvu a jednak hodnotu. Ty budeme realizovat pomocí výčtových typů *Barva* a *Hodnota*. Pro vlastní kartu deklarujeme třídu *Karta*, která má dva neměnné datové atributy hodnota a barva. Dále třída *Karta* obsahuje konstruktor, přístupové metody a metodu *toString()*.

Výčtový typ Hodnota

```
public enum Hodnota {
      Eso, Dve, Tri, Ctyri, Pet, Sest,
    Sedm, Osm, Devet, Deset, Kluk, Dama, Kral
}
```

Třída Karta

```
public class Karta {
   private final Hodnota hodnota; // hodnota karty
   private final Barva barva; // barva karty

   // konstruktor
   public Karta( Hodnota kartaHodnota, Barva kartaBarva ) {
      hodnota = kartaHodnota; // inicializace hodnoty karty
      barva = kartaBarva; // inicializace barvy karty
   }

   public Hodnota getHodnota() {
      return hodnota;
   }

   public Barva getBarva() {
      return barva;
   }

   public String toString() {
     return String.format( "%s  %s", getHodnota(),
         getBarva());
    }
}
```

6.3 Případová studie - karty

Třída *BalicekKaret* obsahuje tři datové atributy. Prvním je pole *balicek* deklarované na 52 prvků a druhým atributem je index tohoto pole. Vidíme, že prvek pole je objektový typ *Karta* (ne pouze primitivní typ nebo řetězec). Třetím datovým atributem, který se využívá pouze pro míchání karet, je proměnná *list*, která bude obsahovat karty.

V konstruktoru se vytvoří všechny karty a uloží se do pole *balicek*. Dále se pole *balicek* načte do proměnné list – řádek 15. Zamíchání karet provádí metoda *shuffle()* v řádku 21. Metoda *tiskSeznamuKaret()* vytiskne zamíchané karty a metoda *tiskKaret()* vytiskne vytvořené karty ještě před zamícháním.

```
import java.util.List;
import java.util.Arrays;
import java.util.Collections;

1 public class BalicekKaret {
2   private Karta[] balicek = new Karta[52];
3   private int count = 0; // pocet karet – index pro pole
4
5   private List< Karta > list; // deklarace seznamu karet
6
7   // vytvoreni balicku karet = konstruktor
8   public BalicekKaret()  {
9      for(Barva barva: Barva.values()) {
10      for(Hodnota hodnota: Hodnota.values()) {
11            balicek[count] = new Karta(hodnota, barva);
12            count++;
13      }
14    }

15   list = Arrays.asList( balicek ); // vytvori List
16   }

17   public Karta[] getBalicek() {
18    return balicek;
19   }

20   public void zamichaniKaret() {
21    Collections.shuffle(list);
22   }

23   public void tiskSeznamuKaret() {
24      // zobrazi 52 karet ve dvou sloupcich
25      for ( int i = 0; i < list.size(); i++ )
26         System.out.printf( "%-20s%s", list.get( i ),
27            ( ( i + 1 ) % 2 == 0 ) ? "\n" : "" );
28   }

29   public void tiskKaret() {
30    for( int i = 0; i < getBalicek().length; i++)
31          System.out.printf( "%-20s%s", balicek[i],
32              ( ( i + 1 ) % 2 == 0 ) ? "\n" : "" );
33    System.out.println();
   }
}
```

Třída *BalicekKaretTest* pak představuje využití. Nejdříve se vytvoří balíček karet, který se vytiskne, pak se zamíchá a opět se vytiskne.

Třída BalicekKaretTest

```
public class BalicekKaretTest {
   public static void main(String[] args) {
       BalicekKaret balicek = new BalicekKaret();
       balicek.tiskKaret();
       balicek.zamichaniKaret();
       balicek.tiskSeznamuKaret();
       }
}
```

Práce s poli jako datovými atributy je prvním krokem k používání seznamů. Často jsou pole opomíjené. Proto uvádíme dva příklady. První ilustruje využívání pole pro uložení číselných hodnot a pole pro využívání řetězců. Druhý příklad naopak ukazuje pole, kde budeme používat výčtové typy. Ty se hodí zejména tam, kde pracujeme s neměnnými hodnotami.

Jaké jsou výhody pole?

Co patří k nevýhodám používání polí?

7. Případová studie – koruny, účet

V této kapitole se dozvíte:

- jak můžeme datový atribut čísla vyjádřit jako třídu,
- jak se změní metody, kdy místo datového atributu deklarujeme třídu,
- na to musíme dávat pozor v metodě toString ve třídě Koruny,
- jak se dá využívat třída *StringBuffer* pro výstup dat.

Po jejím prostudování budete schopni:

- lépe zpracovávat reálné aplikace,
- lépe si uvědomíte vazby a relace mezi třídami,
- rozumět pojmům objekt, třída, budete vědět, jak se třída graficky zobrazuje v diagramu tříd UML, vytvořit a spustit jednoduchý objektově orientovaný program.

Klíčová slova této kapitoly:

případová studie

Pokud jsme pracovali se třídou *Ucet*, pro jednoduchost jsme předpokládali, že datový atribut *stav* je celé číslo. V praxi by to mělo být minimálně reálné číslo. Podívejme se ale na možnost, že datový atribut *stav* by vyjadřoval hodnotu v haléřích a pouze pro výstupní operace by se zobrazoval ve formě korun a haléřů. Ostatně podobně pracuje i kancelářský program Excel, který si pro vlastní potřeby vždy ponechává čísla v nejvyšší přesnosti a pouze pro výstup provádí jejich formátování do požadovaného tvaru.

Abychom tedy zvládli požadované řešení, budeme nejdříve deklarovat třídu *Koruny*, která bude jednak udržovat stav a také bude poskytovat nutné metody. Třída *Koruny* má dva datové atributy a to již zmiňovaný *halere* a třídní (statický) atribut *HALERU_DO_KORUNY*. Tento atribut je pro všechny vytvořené instance stejný a je roven 100. Třída má pro pohodlí dva konstruktory, a to konstruktor s parametrem reálné číslo a konstruktor s parametrem *Koruny*.

Třída Koruny

```
public class Koruny {
   private long halere;
   private static final int HALERU_DO_KORUNY = 100;
```

```
//konstruktory
public Koruny(double castka) {
    this.halere = Math.round(castka * HALERU_DO_KORUNY);
}

public Koruny(Koruny koruny) {
    halere = koruny.getHalere();
}

public long getKoruny() { //vraci počet korun
    return getHalere() / HALERU_DO_KORUNY;
}

public long getHalere() { // vraci castku v halerich
    return halere;
}

public long getHaleru() { // vraci zbyle halere bez korun
    return getHalere() % HALERU_DO_KORUNY;
}

public boolean isZero() {
    return getHalere() == 0;
}

public boolean equals(Object obj) {
    if(!obj.getClass().equals(this.getClass()))
       return false;
    Koruny castka = (Koruny) obj;
    return this.getHalere() == castka.getHalere();
}

public Koruny plus(Koruny castka) {
    return new Koruny(1.0 *(this.getHalere() +
           castka.getHalere()) / HALERU_DO_KORUNY);
}

public Koruny minus(Koruny castka) {
    return new Koruny((double) (this.getHalere() -
             castka.getHalere()) / HALERU_DO_KORUNY);
}

public Koruny nasobeni(double cislo) {
    return new Koruny((double)
      (Math.round(this.getHalere() * cislo))
                        /HALERU_DO_KORUNY);
}

public String toString() {
    StringBuffer vysledek = new StringBuffer("");
    vysledek.append(getKoruny());
    vysledek.append(',');

    long hal = Math.abs(this.getHaleru());
    if(hal == 0) vysledek.append("00");
       else vysledek.append(hal);
    vysledek.append(" Kc");
    return vysledek.toString();
```

```
    }

    public void tisk() {
        System.out.println(this.toString());
    }
}
```

Metoda *equals()* porovnává rovnost dvou objektů typu Koruna. Metody *plus()* a *minus()* vytváří vždy novou instanci typu *Koruny*, která vznikne součtem resp. rozdílem dvou instancí typu *Koruny*. Metoda *nasobeni()* násobí koruny s reálným číslem, např. úrokem.
Třída *KorunyTest* pak ukazuje názorné použití. Nabízí se využít tuto třídu pro jednotkové testování (Unit Testing).

Třída KorunyTest

```
public class KorunyTest {
    public static void main(String[] args) {
        Koruny castka1 = new Koruny(3.5678);
        Koruny castka2 = new Koruny(7.08);

        Koruny vysledek = castka1.plus(castka2);
        vysledek.tisk();

        vysledek = castka1.minus(new Koruny(7.08));
        vysledek.tisk();
        Koruny castka3 = new Koruny(5.94);
        vysledek = castka3.nasobeni(1.28);
        vysledek.tisk();
    }
}
```

Celou aplikaci vidíte na následujícím obrázku. Objekt třídy *Koruny* je částí (kompozice) celku účet a ten je částí (agregace) celku objektu třídy *Student*.

Obr. 7.1 Diagram tříd UML, třídy a jejich vzájemné relace.

Třída *Účet* má dva datové atributy, a to číslo účtu a stav, který je nyní typu *Koruny*. Podívejte se, co se změnilo na třídě *Účet* od doby, kdy stav byl vyjádřen jako primitivní typ int.

Třída Ucet

```
public class Ucet {
   private int cislo;
   private Koruny stav;
```

```
    // Konstruktory tridy Ucet
    public Ucet(){
            this(0, 0);
      }

    public Ucet(int cislo, double castka) {
        this(cislo, new Koruny(castka));
     }

    public Ucet(int cislo, Koruny stav) {
        this.cislo = cislo;
        this.stav = stav;
     }

    public void vlozeni (double castka) {
       // vyvola metodu vlozeni s parametrem Koruny
       this.vlozeni(new Koruny(castka));
     }

    public void vlozeni(Koruny castka) {
        stav = stav.plus(castka);
     }

    public Koruny vyber (double castka) {
       // vyvola metodu vyber s parametrem Koruny
        return this.vyber(new Koruny(castka));
      }

    public Koruny vyber(Koruny castka) {
        stav = stav.minus(castka);
        return getStav();
     }

    public Koruny getStav(){
           return stav;
         }

    public String toString()     {
        return String.format("Ucet cislo: %d stav uctu: %s",
        getCislo(), getStav().toString());
      }

    public void tisk(){
        System.out.println(this.toString());
     }
}
```

Za pozornost jistě také stojí přetížené metody *vlozeni()* a *vyber()*. Vždy první z nich vyvolává druhou metodu s parametrem *Koruny*. Vazba mezi objektem třídy *Účet* a objektem třídy *Koruna* je zvolena jako agregace, protože při vkládání a výběru se do datového atributu stav ukládá nový objekt typu *Koruny*. Pro metody *vyber()* máme požadavek, aby se vracel aktuální stav na účtu.

Třída Student

```
public class Student {
   private String jmeno;
   private Ucet mujUcet;

   public Student(String jmeno, Ucet ucet){
        this.jmeno = jmeno;
        mujUcet = ucet; // agregace
    }

   public Ucet getUcet() {
       return mujUcet;
    }

    public void setUcet(Ucet ucet) {
       mujUcet = ucet;
    }

    public void vlozeni(double castka) {
       // vyvola metodu vlozeni s parametrem Koruny
       this.vlozeni(new Koruny(castka));
    }

    public void vlozeni(Koruny koruny) {
        this.getUcet().vlozeni(koruny);
    }

    public Koruny vyber(double castka) {
       // vyvola metodu vyber s parametrem Koruny
       return this.vyber(new Koruny(castka));
    }

    public Koruny vyber(Koruny koruny) {
        return this.getUcet().vyber(koruny);
    }

   public String toString() {
       return String.format("Jmeno: %s \n%s", getJmeno(),
            getUcet().toString());
    }

   public void tisk() {
       System.out.println(toString());
    }
}
```

Třída Student demonstruje vazbu studenta na účet. Pro vložení a výběr jsou použity přetížené metody podobně, jako tomu bylo ve třídě *Účet*. Rovněž metoda vyběr vrací zbytek na účtu. Praktická aplikace a využití uvedených tříd je ve třídě *StudentTest*. Zde jsou vytvořeny dvě instance třídy *Student* a možnosti aplikace jsou na nich stručně prezentovány.

Třída StudentTest

```
public class StudentTest {
   public static void main(String[] args) {
      Student student1 =
            new Student("Jarek", new Ucet(1, 200));
      student1.tisk();
      student1.vlozeni(300);
      Koruny zbytek = student1.vyber(2300);
      System.out.println("Zbytek na uctu: " + zbytek);

      Ucet ucet = new Ucet(2, new Koruny(500));
      Student student2 = new Student("Alena", ucet);
      student2.tisk();
      zbytek = student2.vyber(400);
      System.out.println("Zbytek na uctu: " + zbytek);
      student1.vlozeni(3500);
      student1.tisk();
   }
}
```

V této kapitole jsme se zabývali případovou studií účet. Tvoří ji třída *Účet*, která je agregovanou součástí třídy *Osoba* a navíc třída *Koruny* je agregovanou součástí třídy *Účet*. Třída *Koruny* rozšiřuje původní datový atribut stav ve třídě *Účet* na samostatnou třídu. Začínáme se třídou *Koruny* a pak je uvedena vazba na ostatní třídy.
Ve řídě Koruny stojí jistě za pozornost metoda *toString()*. Je v ní mimo jiné využita knihovní třída *StringBuffer()*, do jejíž instance se přidávají jednotlivé části tisku pomocí metody *add()*.

Sledujte jak je ve třídách *Účet* a *Osoba* využito přetížených metod pro vložení a výběr. Navíc metoda výběr vrací zůstatek na účtu. Je třeba, aby uživatel mohl zadat částku jako instanci třídy *Koruna*, nebo jako reálné číslo.

8. Návrhové vzory (Design Patterns)

V této kapitole se dozvíte:

- co jsou to návrhové vzory,
- k čemu návrhové vzory slouží a jak se dokumentují,
- jak se implementují v konkrétní aplikaci.

Po jejím prostudování budete schopni:

- využívat návrhový vzor přepravka,
- využívat návrhový vzor singleton – jedináček.

Klíčová slova této kapitoly:

návrhový vzor, singleton, přepravka

Návrhové vzory jsou důležitou součástí vytváření programového řešení hned z několika důvodů, a proto s nimi začínáme již zde a teď.

Návrhové vzory jsou doporučené postupy řešení často se vyskytujících úloh. Mohli bychom je přirovnat k vzorečkům z matematiky nebo fyziky. Například řešení kvadratické rovnice. Když známe vzoreček, řešení není žádným problémem. V návrhových vzorech však na rozdíl od matematických vzorců budeme pracovat s rozhraními, třídami a objekty. V návrhovém vzoru neexistují konkrétní třídy ani konkrétní objekty (případně jiné prvky modelů), ale vyskytují se v nich jejich proměnné (proměnné tříd, proměnné objektů atd.) Tyto proměnné jsou chápány jako role, za které lze dosadit konkrétní třídy, konkrétní objekty resp. jiné konkrétní prvky našeho řešeného problému, a tak dostáváme řešení požadované situace.

První z výhod používání návrhových vzorů je snížení pravděpodobnosti výskytu chyb při použití návrhových vzorů, než při vlastním řešení. Navíc většinou návrhové vzory počítají s budoucím možným rozšířením, takže i zde usnadňují práci.

Další nezanedbatelnou výhodou, jakýmsi vedlejším efektem (side-effect) je, že při studiu a používání návrhových vzorů lépe zvládneme zásady objektově orientovaného přístupu. Jednou z nich je využívání tzv. interfacového přístupu k řešení problému (přístup pomocí rozhraní). K problematice rozhraní se dostaneme až v dalších kapitolách. (Návrhové vzory uváděné v této kapitole jsou úvodní jednoduché vzory). Kromě přístupu přes rozhraní je v návrhových vzorech uplatněn také polymorfismus. Dá se dokonce říci, že nosným pilířem návrhových vzorů je využití polymorfismu v objektově orientovaném přístupu.

Obecně lze vzor považovat za opětovně použitelný obecný návod k řešení problému, který se neustále opakuje v různých obdobách a v různých situacích.

Na chápání a na použití návrhových vzorů je obtížná právě jejich abstrakce. Samotný vzor totiž nepopisuje nějakou konkrétní situaci k řešení, ale popisuje zobecnění všech takových situací.

A teď již konkrétně k jednotlivým vzorům.

8.1 Přepravka

Problém:
Sloučení několika samostatných informací do jednoho celku. Například víme, že metody používané v deklaraci tříd vracejí pouze jedno hodnotu, nebo pouze jeden objekt. Podobají se takto funkcím. Co ale udělat, když bychom potřebovali, aby metoda vrátila více hodnot resp. více objektů?

Kontext:
Vrácení několika hodnot současně z dané metody.

Řešení:
Deklarace třídy typu Přepravka, jejíž datové atributy budou využity k uložení požadovaných samostatných informací. Návratovým objektem z metody pak může být právě objekt typu přepravka.
Implementace:
Datové atributy třídy Přepravka se deklarují jako **veřejné (public),** aby k nim byl snadnější přístup (nebylo nutné používat přístupových resp. modifikačních metod). Po přenesení hodnot z přepravky se objekt přepravky většinou „zahodí“, dále nepoužívá.

Nejdříve několik úvodních vysvětlení.
V následujícím příkladu máme v balíčku (package) messenger uvedeno několik tříd demonstrující návrhový vzor přepravka bez specifikace public. Pro takové třídy platí, že:

- jsou viditelné pouze v rámci svého balíčku,
- nemusí mít svůj zdrojový kód uložen v souboru se stejným jménem,
- mohou být uloženy v souboru s jinými třídami,
- název souboru je podle třídy se specifikací public, v našem případě je tedy vše uloženo v souboru „PrepravkaDemo.java“.

V tomto návrhovém vzoru se často vyskytuje tzv. kopírovací konstruktor (copy constructor). Kopírovací konstruktor vytváří novou

instanci jako kopii instance předané jako parametr. Datové atributy předávané instance se zkopírují do nové instance.

Vlastní přepravku vytváří třída *Point*. Její datové atributy jsou deklarované jako public – nemusí se tedy při práci s nimi používat přístupové a modifikační metody (get a set).

Třída Bod (jako přepravka)

```
// prepravka
public class Bod {
  public int x, y, z;
  public Bod(int x, int y, int z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  // kopirovaci konstruktor
  public Bod(Bod p) {
    this.x = p.x;
    this.y = p.y;
    this.z = p.z;
  }

  public String toString() {
    return "x: " + x + " y: " + y + " z: " + z;
  }
}
```

Třída Vektor

```
public class Vektor {
   private int velikost, smer;
   public Vektor(int velikost, int smer) {
      this.velikost = velikost;
      this.smer = smer;
      }
   public int getVelikost() {
        return velikost;
   }
   public int getSmer() {
        return smer;
   }
}
```

Třída Plocha

```
public class Plocha {
   public static Bod translate(Bod p, Vektor v) {
      p = new Bod(p); // nemodifikuje objekt v parametru
      // vykonani smyslenych operaci:
       p.x = p.x + v.getVelikost();
       p.y = p.y + v.getSmer();
       p.z = p.z + v.getVelikost() * v.getSmer();
   return p;
```

```
   }
}
```

Třída Přepravka

```
public class Prepravka {
  public static void main(String[] args) {
    Bod p1 = new Bod(1, 2, 3);
    Bod p2 = Plocha.translate(p1, new Vektor(11, 47));
    String vysledek = "p1: " + p1 + "\np2: " + p2;
    System.out.println(vysledek);
  }
}
```

Vlastní využití návrhového vzoru přepravka je demonstrováno ve statické metodě translate třídy Plocha, která vrací právě tři jednoduché hodnoty typu int jako objekt třídy *Bod*. Výraz:

```
p = new Bod(p);
```

vytvoří novou instanci *p* třídy *Bod*, do které se zkopírují původní datové atributy instance *p* předávané do této metody jako parametr. Stejné pojmenování objektů se používá z toho důvodu, abychom neupravili datové atributy původního objektu předávaného jako parametr.
Výsledkem je tisk objektů *p1* a *p2*, které se navzájem liší.

8.2 Singleton - jedináček

Problém
Zabezpečit, aby třída měla pouze jednu instanci (objekt) globálně viditelnou ze všech částí programu.

Kontext
V mnoha případech je třeba, aby ze třídy vznikla pouze jedna sdílená instance pro celý program.

Řešení
Implementace singletonu je celá řada. Mají však společné to, že definují konstruktor třídy, která má mít pouze jednu instanci jako soukromý (private). Tím je zabezpečeno, že se k tomuto konstruktoru dostaneme pouze prostřednictvím jiné další metody, která bude veřejná (public).

Modifikátor *final* způsobí, že třída *Singleton* se již dále nemůže rozvíjet, nemůže mít další podtřídy. Je to z toho důvodu, aby eventuálně v podtřídě nemohl existovat konstruktor. Podstatné je, že deklarovaná třída má jeden konstruktor, který je deklarovaný jako private (soukromý). Není tedy přístupný přímo. Přímo v deklaraci třídy

se vytvoří nová instance (objekt) s označením *jedinacek*, který je přístupný prostřednictvím metody *getInstance()*. Tato metoda je statická, což umožňuje vytvářet instance s využitím třídy:

```
1 public final class Ucet {
2   private int cislo;
3   private Koruny stav;
4   private static Ucet jedinacek;
5
6   // Konstruktory tridy Ucet
7   private Ucet(){
8         this(0, 0);
9     }

10   private Ucet(int cislo, double castka) {
11       this(cislo, new Koruny(castka));
12    }

13   private Ucet(int cislo, Koruny stav) {
14       this.cislo = cislo;
15       this.stav = stav;
16    }

17   public static Ucet getInstance() {
18    if(jedinacek == null) jedinacek = new Ucet(1, 200);
19    return jedinacek;
   }
```

Třída *Ucet* je vytvořena jako singleton, jedináček. Bude existovat tedy pouze jedna instance dané třídy. K tomu, abychom z dané třídy udělali třídu singleton, musíme provést následující úpravy:

- deklarujeme třídní (statický) datový atribut typu té dané třídy – řádek 4,
- deklarujeme všechny kosntruktory dané třídy jako private – řádky 7, 10, 13,
- deklarujeme danou třídu jako final, tím znemožníme vytváření podtříd, které by už nebyly singletony,
- deklarujeme metodu např. *getInstance()*, která vrací instanci dané třídy a která funguje podobně jako kód na řádcích 17-19.

```
1  public class Student {
2  private String jmeno;
3  //private Ucet mujUcet;
4  // druha moznost pouziti vzoru
5  //Ucet ucet = Ucet.getInstance();
6
7  public Student(String jmeno){
8       this.jmeno = jmeno;
9      // mujUcet = ucet; // agregace
10   }
```

```
11  public String getJmeno() {
12      return jmeno;
13   }
14
15   public void vlozeni(double castka) {
16    // vyvola metodu vlozeni s parametrem Koruny
17      this.vlozeni(new Koruny(castka));
18   }

19   public void vlozeni(Koruny koruny) {
20       //this.getUcet().vlozeni(koruny);
21    Ucet.getInstance().vlozeni(koruny);
22   }

23  public Koruny vyber(double castka) {
24    // vyvola metodu vyber s parametrem Koruny
25     return this.vyber(new Koruny(castka));
26  }

27  public Koruny vyber(Koruny koruny) {
28      //return this.getUcet().vyber(koruny);
29      return Ucet.getInstance().vyber(koruny);
30  }

31 public String toString() {
31     return String.format("Jmeno: %s \n%s", getJmeno(),
32          //getUcet().toString());
33                Ucet.getInstance().toString());
34  }

35 public void tisk() {
36     System.out.println(toString());
   }
}
```

Třída *Student* je třída, jejíž instance budou spolupracovat s instancí singletonu třídy *Ucet*. Jak je vidět z kódu, třída *Student* nemá žádný datový atribut s vazbou na instance třídy *Ucet*. Tato možnost je zakomentovaná. Pokud se potřebujeme ve třídě Student odkazovat na Ucet, máme k tomu dvě možnosti:

- jako datový atribut si deklarujeme proměnnou třídy Ucet, které vyvoláme třídní metodu *getInstance()* viz. řádek 5; pak nám stačí se kdekoli ve třídě *Student* na instanci třídy *Ucet* odkazovat prostřednictvím proměnné *ucet*,

- v každé metodě, kde potřebujeme odkaz na instanci třídy *Ucet* použijeme konstrukci: *Ucet.getInstance().toString();* konstrukce *Ucet.getInstance()* vrátí singleton.

Upozornění – je třeba vybrat jen jednu možnost! V kódu programu je první možnost zakomentovaná, druhá je účinná.

Třída StudentTest

```
public class StudentTest {
  public static void main(String[] args) {
      Student student1 = new Student("Jarek");
      student1.tisk();
      student1.vlozeni(300);
      Koruny zbytek = student1.vyber(2300);
      System.out.println("Zbytek na uctu: " + zbytek);

      Student student2 = new Student("Alena");
      student2.tisk();
      zbytek = student2.vyber(400);
      System.out.println("Zbytek na uctu: " + zbytek);
      student1.vlozeni(3500);
      student1.tisk();
    }
}
```

V uvedeném příkladu se vytvoří dvě instance, a to *student1* a *stundent2*, které obě ukazují na stejný objekt. Vkládají a vybírají ze společného účtu. Můžete se o tom přesvědčit.

Návrhové vzory hrají stále důležitější roli v objektově orientovaném programování. Mnozí experti v této oblasti tvrdí, že prostřednictvím návrhových vzorů pochopili daleko lépe principy OOP. Uvedené dva návrhové vzory jsou pouze úvod do uvedené oblasti.

Návrhový vzor přepravka řeší problematiku návratu dvou a více objektů z dané metody.
Návrhový vzor sigleton (jedináček) zabezpečí vytvoření pouze jedné instance od dané třídy. Obecně ale tento návrhový vzor zabezpečuje tvorbu omezeného počtu instancí dané třídy, tedy ne jen jediné instance.

Co je charakteristické pro návrhové vzory?

4.1 Jak se zabezpečí, aby se k metodě třídy dostaly všechny objekty (instance) této třídy?

4.2 Co způsobí, když je konstruktor deklarovaný s modifikátorem private?

4.1 Metoda se označí klíčovým slovem static.

4.2 K tomuto konstruktoru není možné se dostat přímo, proto to řešíme prostřednictvím jiných metod, deklarovaných jako public – viz metoda *getInstance( )*.

9 Balíčky, dědičnost, třída Object

V této kapitole se dozvíte:

- proč se zavádějí a k čemu se používají balíčku (name spaces – jmenné prostory)
- co dědí podtřída od nadtřídy,
- jak funguje překrývání metod,
- jak fungují konstruktory při využívání dědičnosti,
- jakou má strukturu třída Object.

Po jejím prostudování budete schopni:

- využívat balíčky,
- využívat dědičnosti při návrhu aplikací,
- uvědomíte si, co poskytuje třída Object.

Klíčová slova této kapitoly:

balíček, dědičnost, třída Object

9.1 Balíčky

Jedním z problémů větších (velkých) programových celků je kolize jmen použitých identifikátorů. Což v podstatě znamená, abychom my při vlastní tvorbě již nepoužili identifikátor použitý někde v knihovně, nebo identifikátor, který používáme sami, nebo používá spolupracující kolega.
Java řeší kolizní problém jmen zavedením konceptu **namespace** (prostor jmen, nebo jmenný prostor), který se v Javě nazývá **package** česky paket nebo balíček. Znamená to, že skupiny příbuzných tříd jsou uloženy do jednoho balíčku. Balíček fyzicky představuje adresář na disku a bývá v různých vývojových prostředích realizován trošku odlišně, principiálně však stejně. Prostředí BlueJ je trochu omezenější při práci s balíčky než prostředí Elipse.

Pro třídy v daném balíčku platí, že jejich identifikátory (názvy tříd) musí být jedinečné. K tomu, aby příslušná třída patřila do daného balíčku, musí být příslušná třída uložena v patřičném adresáři (podadresáři) reprezentující daný balíček, ale také musí programátor explicitně potvrdit, že tam patří - zdrojový text musí začínat příkazem:

package název_balíčku;

V následujícím příkladu je uveden balíček směnárna. Pro název balíčku se používají malá písmena. Kromě toho se dají balíčky uspořádat hierarchicky, podobně jako adresáře souborů. Je to výhodné zejména pro ukládání jiných verzí, dále pak se do balíčků mohou ukládat různé typy souborů např. zdrojový, zkompilovaný, spakovaný atd.

Názvy tříd

Rozmístění tříd do balíčků ovlivní jejich názvy. Kdybychom jako doposud používali pouze názvy tříd, vystavovali bychom se nebezpečí stejných názvů tříd z různých balíčků. Proto se název třídy skládá z názvu balíčku (balíčků) a názvu třídy odděleného tečkami. Např.

smenarna.SmenarnaBezPoplatku
c8.inner.Dispatcher

Třída Dispatcher je umístěna v balíčku *inner*, který je součástí balíčku *c8*.

Balíčky a příkaz import

Aby byl překladač ochoten pracovat s deklarovanými třídami v různých balíčcích, musíme použít příkaz import a deklarovat patřičnou cestu k balíčkům. Např.:

import c8.inner.Dispatcher;

je příkaz pro importování pouze třídy Dispatcher z uvedených balíčků. Naopak, pokud uvedeme pouze:

import c8.inner. ;*

budou se importovat všechny třídy uvedené v cestě balíčků. Jak jste si jistě všimli, podobným způsobem se importují i všechny požadované třídy z javovských knihoven.

9.2 Dědičnost

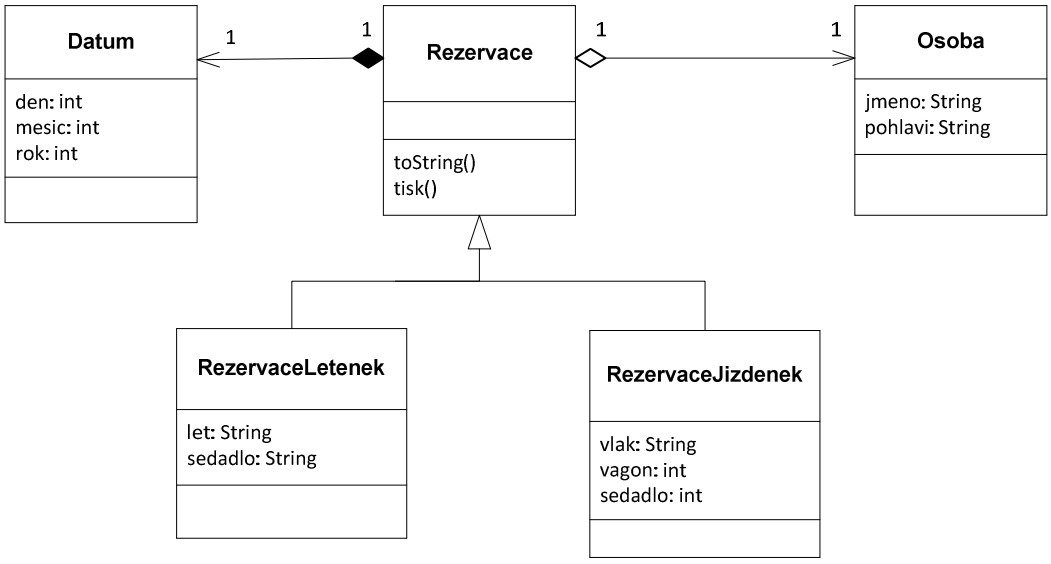
Dědičnost v objektově orientovaném přístupu představuje mechanismus vytváření tříd a podtříd. Podtřída (subclass) dále specializuje svoji nadtřídu a to formou dalších datových atributů, nebo dalších resp. specializovanějších metod. Další metody jednoduše představují metody, které nadtřída nedeklaruje. Specializovanější metody mají stejný název jako metody nadtřídy. V tomto případě dojde buď k úplnému „překrytí“ (lepší termín zastínění, anglický používaný termín – overriding) metody nadtřídy, nebo může být využit kód nadtřídy, který v podtřídě rozšíříme (specializujeme) s využitím pseudoproměnné super. Pseudoproměnná **super** odkazuje na nadtřídu. Obě uvedené možnosti si ukážeme na příkladech.

Postup směrem od nadtřídy k podtřídě se nazývá specializace, opačný postup pak generalizace – zevšeobecnění.

Vztahy podtřídy a nadtřídy spolu s dalšími pravidly, která při dědičnosti platí, si vysvětlíme na příkladu rezervace. *Rezervace* je třída, která deklaruje základní datové atributy a operace pro vlakovou a leteckou rezervaci. Její datové atributy tvoří datum, reprezentované instanci třídy Calendar z balíčku java.util. Mezi objekty *Rezervace* a *Datum* je spojení pomocí kompozice – tato asociace se realizuje v konstruktoru.

Dalším datovým atributem deklarovaným ve třídě *Rezervace* je objekt třídy *Osoba*, reprezentujícího zákazníka. Mezi objekty tříd *Rezervace* a *Osoba* je spojení pomocí agregace. Tato asociace se realizuje pomocí metody *setZakaznika()* až v běhu programu.

Podtřídami třídy *Rezervace* jsou třídy *RezervaceLetenek* a *RezervaceJizdenek*. Vše je uvedeno v UML diagramu tříd.



Obr. 7.1 Diagram tříd UML

Podstatou dědičnosti (vytváření hierarchii tříd) je, že objekty *dědí* od nadřazených objektů strukturu lokálních datových atributů (ne obsah) a operace které jsou nad nimi deklarovány. Vzniká tak hierarchie objektů, přičemž nejobecnější objekty mají jen relativně málo společných vlastností (lokální data a operace nad nimi). Objekty, které jsou v hierarchii níže, mohou být postupně odlišovány od obecných.

Ukrývání informací v hierarchii tříd

Na ukrývání informací používá jazyk Java modifikátory private, public a protected. Nejdříve probereme modifikátory public a private. Doposud jsme většinou popisovali datové atributy modifikátorem private (s výjimkou návrhového vzoru přepravka) a metody většinou modifikátorem public. V rámci jedné třídy jsme si mohli i dovolit nepoužívat přístupových a modifikačních metod a upravovat datové atributy přímo (bez těchto metod). Pokud se však z podtřídy odkazujeme na datové atributy nadtřídy, jinak než s využitím přístupových a modifikačních metod to nejde. Je to z toho důvodu, že právě datové atributy využívají modifikátor *private* a metody modifikátor *public*.

Byla by zde ještě jedna alternativa, využívat pro datové atributy modifikátor protected, který zabezpečuje „přímý přístup“ (bez použití přístupových a modifikačních metod) k datovým atributům z podtřídy do nadtřídy. Jeho nevýhodou je však to, že datové atributy s modifikátorem protected jsou však „přímo“ přístupné také všem třídám v daném balíčku. Z jiného pohledu je však výhodné používat pouze metod k přístupu k datovým atributům, protože takto je zabezpečen pouze metodami definovaný přístup – což patří k základním principům objektově orientovaného přístupu.

Nyní uvedeme výpisy podstatných tříd nebo jejich částí. Třídu *Osoba* neuvádíme.

```
import java.util.Calendar;
public class Datum {
   private int den;   // 1-31
   private int mesic; // 1-12
   private int rok;
   private Calendar kalendar = Calendar.getInstance();
   // kalendar.get(1) = rok, kalendar.get(2) +1 = mesic,
   // kalendar.get(5) = den

   //konstruktory
   public Datum( int den,int mesic,  int rok)  {
      this.mesic = mesicKontrola( mesic );
      this.rok = kontrolaRok(rok);
      this.den = denKontrola( den);
   }

   public Datum() {
      mesic = kalendar.get(2) + 1;
      den = kalendar.get(5);
      rok = kalendar.get(1);
    }
```

```
public Datum(int den) {
   mesic = kalendar.get(2) + 1;
   this.den = denKontrola(den);
   rok = kalendar.get(1);
 }

public Datum(int den, int mesic) {
   this.mesic = mesicKontrola(mesic);
   this.den = denKontrola(den);
   rok = kalendar.get(1);
 }

public Datum(int den, String mesic, int rok) {
    this.mesic = mesicKontrola(mesic);
    this.den = denKontrola(den);
    this.rok = kontrolaRok(rok);
 }

public Datum(int den, Mesic mesic, int rok) {
    this.mesic = mesic.ordinal() + 1;
    //System.out.println("mesic: " + this.mesic);
    this.den = denKontrola(den);
    this.rok = rok;
 }

// kopirovaci konstruktor
public Datum(Datum d) {
    mesic = d.getMesic();
    den = d.getDen();
    rok = d.getRok();
 }

// pristupove a modifikacni metody . . .
private int mesicKontrola( int mesic) {
   if ( mesic > 0 && mesic <= 12 )
      return mesic;
   else
   {
      System.out.printf(
         "Neplatny mesic (%d) nastaveni na 1.", mesic );
      return 1;
   }
}

private int mesicKontrola(String mesic) {
   String[] mesice =
   {"leden", "unor", "brezen", "duben", "kveten", "cerven",
        "cervenec", "srpen", "zari", "rijen", "listopad",
                                    "prosinec"};
   for(int i = 0; i < mesice.length; i++)
       if(mesic == mesice[i]) return i+1;
   System.out.printf("neplatny mesic (%s)“ +
         “ nastaveni na leden ", mesic);
   return 1;
 }
```

```
   private int denKontrola( int den ) {
      int dnuVMesici[] =
       { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
                                                 };

      // kontrola rozsahu dne v mesici
      if ( den > 0 && den <= dnuVMesici[ getMesic() ] )
         return den;

      // kontrola prestupny rok
      if ( getMesic() == 2 && getDen() == 29 && ( getRok() %
         400 == 0 ||
           ( getRok() % 4 == 0 && getRok() % 100 != 0 ) ) )
         return den;

      System.out.printf( "Neplatny den (%d) nastaveni na 1.",
               den );
      return 1;
   }

   public int kontrolaRok(int r) {
       if(r>=1900 && r <= 2100) return r;
         else
          return kalendar.get(1);
   }

   public String toString() {
      return String.format( "%d/%d/%d", getDen(), getMesic(),
                                                   getRok());
   }

   public void tisk(){
       System.out.println(toString());
    }
}
```

Výčtový typ Mesic

```
public enum Mesic {
    LEDEN, UNOR, BREZEN, DUBEN, KVETEN, CERVEN, CERVENEC,
    SRPEN, ZARI, RIJEN, LISTOPAD, PROSINEC
}
```

Třída Rezervace

```
public class Rezervace {
   Datum datum;
   Osoba zakaznik;
   public Rezervace(Datum datum){
      this.datum = datum;
   }

   public void setZakaznik(Osoba o){
      zakaznik = osoba;
   }
```

```
    public Osoba getZakaznik(){
        return zakaznik;
    }

    public String toString(){
        return String.format("Zakaznik: %sDatum: %s",
                    zakaznik.toString(), datum.toString());
    }

    public void tisk(){
        System.out.print(this.toString());
    }
}
```

Třída RezervaceLetenek

```
public class RezervaceLetenek extends Rezervace {
    private String let;
    private String sedadlo;

    public RezervaceLetenek(Datum d, String l, String s){
        super(d); let = l; sedadlo = s;
    }

    public String getLet(){
        return let;
    }

    public String getSedadlo() {
        return sedadlo;
    }

    public String toString() {
        return String.format("%s \nLet: %s sedadlo: %s\n",
               super.toString(),getLet(),getSedadlo());
    }
}
```

Třída RezervaceJizdenek

```
public class RezervaceJizdenek extends Rezervace {
    private String vlak;
    private int vagon;
    private int sedadlo;

    public RezervaceJizdenek(Datum d, String v, int vg, int s){
              super(d); vlak = v;
              vagon = vg; sedadlo = s;
    }

    public String getVlak(){
        return vlak;
    }
```

```
   public int getVagon(){
      return vagon;
   }

   public int getSedadlo(){
      return sedadlo;
   }

   public String toString(){
      return String.format("%s\nVlak: %s vagon: %4d,“ +
            “ sedadlo: %3d\n",
                  super.toString(),vlak, vagon, sedadlo);
   }
}
```

Jak je vidět z výpisu pro vyjádření vztahu podtřída & nadtřída se používá klíčové slovo **extends** (rozšiřuje).

Následující zdrojový výpis je třída *RezervaceTest* na které si ukážeme praktické využití dědičnosti.

Třída RezervaceTest

```
public class RezervaceTest {
   public static void main(String[] args) {
1     RezervaceJizdenek jizdenka =
         new RezervaceJizdenek(new Datum(25),
2                              "Pendolino",506,28);
3
4     Osoba osoba = new Osoba("Adam","muz");
5           jizdenka.setZakaznik(osoba);
6
7     RezervaceLetenek letenka = new RezervaceLetenek(
8            new Datum(16, Mesic.UNOR, 2014),
                                          "Ok 225","14A");
9     osoba = new Osoba("Alena","zena");
10    letenka.setZakaznik(osoba);
11    // tisk rezervace jizdenky
12    jizdenka.tisk();
13    // tisk rezervace letenky
14    letenka.tisk();
15
16    System.out.println("Let: "+letenka.getLet());
17
18    //chybny pristup - informace z nadtridy pouze
19    // prostrednictvim pristupovych metod
20    //System.out.println("Jmeno zakaznika: " +
      //         letenka.zakaznik.jmeno);
21
22    System.out.println("Jmeno klienta: " +
23                     letenka.getZakaznik().getJmeno()+"\n");
24    // kvalifikace promenne rezervace
25    Rezervace rezervace;
26    rezervace = jizdenka; // pretypovani na predka
27    rezervace.tisk();
```

```
28      rezervace = letenka; // pretypovani na predka
29      rezervace.tisk();
30      System.out.println("Jmeno: "
           + rezervace.getZakaznik().getJmeno()+
31      " pohlavi: "+rezervace.getZakaznik().getPohlavi());
32      //metoha getLet() není definovana pro tridu Reservace;
33      //System.out.println("Let: "+rezervace.getLet());
34
35      RezervaceLetenek letenkaX;
36      letenkaX = (RezervaceLetenek) rezervace;
37      System.out.println("Let: "+letenkaX.getLet());
        }
}
```

Pro snadnější pochopení a vysvětlení jsou řádky výpisu očíslovány. Nyní ještě uvedeme výpis programu - to co se vypíše na konzolu.

```
Zakaznik:  jmeno: Adam pohlavi: muz
Datum: 25/7/2013
Vlak: Pendolino vagon:  506, sedadlo:  28

Zakaznik:  jmeno: Alena pohlavi: zena
Datum: 16/2/2014
Let: Ok 225 sedadlo: 14A

Let: Ok 225
Jmeno klienta: Alena

Zakaznik:  jmeno: Adam pohlavi: muz
Datum: 25/7/2013
Vlak: Pendolino vagon:  506, sedadlo:  28

Zakaznik:  jmeno: Alena pohlavi: zena
Datum: 16/2/2014
Let: Ok 225 sedadlo: 14A

Jmeno: Alena pohlavi: zena
Let: Ok 225
```

Poznámky k programu *RezervaceTest*:

1.-2. řádek - vytvoření objektu v třídy *RezervaceJizdenek*.
4. řádek – vytvoření objektu třídy *Osoba*
5. řádek – přiřazení odkazu objektu třídy *Osoba* do objektu třídy *RezervaceJizdenek*

13. resp. 15. řádek tisk objektů třídy *RezervaceJizdenek* resp. *RezervaceLetenek*

16. řádek – tisk datového atributu let objektu *letenka* třídy *RezervaceLetenek*. Používáme přístupovou metodu *getLet*(), protože objekt *letenka* je kvalifikovaný ve třídě *RezervaceLetenek*.

20. řádek – chyba - je třeba použít přístupové metody jak pro *zákazníka*, tak i pro jeho *jméno*, vše je správně uvedeno v řádku 22-23

25. řádek – kvalifikace objektu *rezervace* třídy *Rezervace*
Protože třída *Rezervace* je všeobecnější, než *RezervaceLetenek* resp. *RezervaceJizdenek*, je možné, aby odkazovala na objekty svých podtříd. Viz řádek 26, kdy odkazuje na objekt třídy *RezervaceJizdenek* a řádek 28, kdy odkazuje na objekt třídy *RezervaceLetenek*. Obráceně to nejde (kvalifikujeme proměnnou jako objekt např. třídy *RezervaceJizdenek* a snažili bychom do tohoto objektu přiřadit objekt třídy *Rezervace* – v uvedeném příkladě nemáme vytvořený objekt třídy *Rezervace*, proto můžete vyzkoušet sami).

27. a 29. řádek demonstrují princip pozdní vazby. Objekt *rezervace* kvalifikovaný jako *Rezervace* za běhu programu ukazuje na objekty svých podtříd a metoda tisk ukazuje, že se vytisknou požadované atributy objektů tříd, na které byla proměnná *rezervace* „přesměrována“. Brzká vazba říká, že kód metody je znám již v době překladu programu. Naproti tomu pozdní vazba – náš nynější případ říká, že kód metody je určen až za běhu programu.

Je to dáno tím, že proměnná kvalifikovaná jako proměnná dané nadtřídy může během běhu programu ukazovat na objekty svých podtříd. V případě, že dostane zprávu něco provést, se nejdříve musí zjistit, na který objekt daná proměnná aktuálně ukazuje a teprve podle toho se vybere metoda patřící ke třídě daného objektu.

Další důležité pravidlo, které patří k dědičnosti, je uvedeno v řádcích 30 a 33. V řádku 30 program vypisuje u proměnné *rezervace* (kvalifikována jako *Rezervace*, ukazuje na objekt třídy *RezervaceLetenek*) datové atributy jméno a pohlaví. vše proběhne bez problému. Když ale chceme na řádku 33 vypsat stejným způsobem číslo letu, překladač ohlásí chybu – metoda *getLet*() není deklarovaná pro typ *Rezervace*. Z toho plyne závěr, že proměnná má přístup prostřednictvím přístupových a modifikačních metod pouze k těm datovým atributům, které jsou deklarovány ve třídě, pro kterou **byla daná proměnná kvalifikována.**

Konstruktory a dědičnost

Jak je vidět z výpisu programů, prvním příkazem v konstruktoru podtřídy je volání konstruktoru bezprostřední nadtřídy pomocí pseudoproměnné super (podobně jako když se v rámci jedné třídy odkazujeme na jiné konstruktory s využitím pseudoproměnné this). Samozřejmě rozlišujeme, zda je konstruktor nadtřídy bezparametrický – *super()*, nebo vyžaduje zaslání nějakých parametrů – *super(int hodnota1, String hodnota2, long hodnota3 ..).*

Za zmínku stojí, že v konstruktoru podtřídy musíme zadat všechny parametry, tedy i parametry všech nadtříd. Např. v konstruktoru *RezervaceJizdenek* musíme kromě vlaku, vagonu a sedadla zadat samozřejmě první parametr datum. Pořadí je dané pořadím deklarace.

9.3 Překrývání (zastiňování) metod

Metoda *toString()* je příkladem metody, kterou si deklaruje každá třída. Pokud daná třída deklaruje metodu stejného jména a stejných parametrů jako metoda nadtřídy, bude metoda nadtřídy v podtřídě zastíněna. Tedy bude platit metoda podtřídy. Ale konstrukce *super.toString()* znamená vyvolání metody *toString()* z bezprostřední nadtřídy. Pseudoproměnná super odkazuje na nejbližší nadtřídu, podobně jako pseudoproměnná *this* odkazuje na sebe sama, jako na třídu.

Když se blíže podíváme na její konstrukci tak zjistíme, že používá konstrukci *super.toString*(). *super* je odkaz na bezprostřední nadtřídu a toString() je metoda. To znamená, že metoda *toString*() je vytvořena tak, že nejdříve zajde do nejvyšší třídy v hierarchii tříd a do řetězcové proměnné uloží datové atributy této třídy. Pak sestoupí o úroveň níže a celý postup opakuje, až se dostane do aktuální třídy, kde přidá do řetězcové proměnné datové atributy aktuální třídy.

Metoda *tisk*() se vyskytuje pouze ve třídě Rezervace. To znamená, že pokud instance nějaké podtřídy třídy Rezervace vyvolá metodu *tisk()*, tak se bude nejdříve hledat v konkrétní podtřídě. Tam se ale nenajde a proto se bude s hledáním pokračovat v nadtřídě tedy v Rezervaci. Tam je deklarovaná metoda tisk(), a proto se použije.

9.4 Třída Object

Třída **Object** představuje nejvyšší třídu v hierarchii tříd. Protože je umístěna v balíčku java.lang (implicitně importován) je celý název *java.lang.Object*. Všechny námi doposud definované třídy jsou implicitně podtřídami této třídy. Třídu Object můžeme uvést i explicitně. Např.
public class *Auto* extends *Object* { ... }

Třída *Object* poskytuje svým podtřídám pouze metody, je to vlastně abstraktní třída (viz dále). V následující tabulce jsou uvedeny metody třídy Object a jejich stručný význam. Doposud jsme se setkali s metodami *toString*() a *getClass*().

Tabulka 9.1 Metody třídy Object

Metoda	Popis
clone	metoda protected, bez argumentů, vytváří kopii objektu, na který je zavolána; v dané třídě se předpokládá redefinování této metody jako public, která by měla implementovat rozhraní Clonable (balíček java.lang). Implicitní implementace této metody provádí tzv. shallow copy.
equals	metoda porovnává dva objekty a vrací true v případě rovnosti a false při nerovnosti; object1.equals(object2); metoda předpokládá redefinici ve třídách použití
finalize	metoda protected, je volána garbage collectorem ke konečnému uklízení, nemá parametry, vrací void, zřídka kdy používaná
getClass	metoda vrací informace o dané třídě, jméno třídy dostaneme pomocí metody getName()
hashCode	hashovací tabulka je datová struktura, která přiřazuje objekt klíč k objektu hodnota, hashCode je hodnota vrácená metodou hashCode, která se používá ke stanovení místa, kde bude uložena odpovídající hodnota
notify notifyAll wait	metody využívané pro multithreading (aktivní objekty)
toString	tato metoda vrací textovou (řetězcovou) reprezentaci objektu

Tato kapitova se zaměřuje na dědičnost, ale kromě dědičnosti vysvětluje smysl a význam zavedení balíčků.
Balíčky řeší problém kolize názvů proměnných. V balíčku se sdružují třídy řešící podobný problém. Balíčky se dají vytvářet hierarchicky.
Dědičnost patří k základním charakteristikám OOP. K tomu je třeba také znát strukturu třídy Object.

Co platí o proměnnou, která je deklarovaná na úrovni nadtřídy?

Tato proměnná může ukazovat na instance té konkrétní nadtřídy a na instance všech podtříd dané nadtřídy.

Vysvětlete na krátném příkladě co znamená, že proměnná nadtřídy může ukazovat na instance všech svých podtříd.

Jak se objekt podtřídy dostane k datovému atributu nadtřídy, který má modifikátor private? Jak probíhá překrývání (zastiňování) metod? K čemu a jak se používá pseudoproměnná *super*?

Při vytváření objektu podtřídy je nutné zadat:
a) pouze hodnoty datových atributů dané podtřídy?
b) hodnoty datových atrubutů jak dané podtřídy, tak i všech nadtříd?

7.1 Proměnná kvalifikovaná v nadtřídě a přetypovaná na objekt podtřídy se může dostat ke kterým datovým atributům?
7.2 Jak by bylo možné doplnit do příkladu Rezervace ještě rezervaci na autobusy?

7.1 Dostane se pouze k datovým atributům třídy, pro niž byla kvalifikována..
7.2. Do hierarchie tříd by se doplnila třída AutobusovaRezervace na úroveň letecké a vlakové rezervace.

Jak se objekt podtřídy dostane k datovému atributu nadtřídy, který má modifikátor private? Jak probíhá překrývání (zastiňování) metod?
K čemu a jak se používá pseudoproměnná super?

Hierarchie tříd (dědičnost) patří k základním charakteristikám objektově orientovaného programování. Důležité je správně navrhnout hierarchii tříd a příslušnost jednotlivých datových atributů k odpovídajícím třídám.

10 Polymorfismus, abstraktní třída, rozhraní

V této kapitole se dozvíte:

- jak se prakticky využívá polymorfismus,
- co je to abstraktní třída, jakou má strukturu a čím se liší od normální třídy,
- jako strukturu má rozhraní a k čemu se používá.

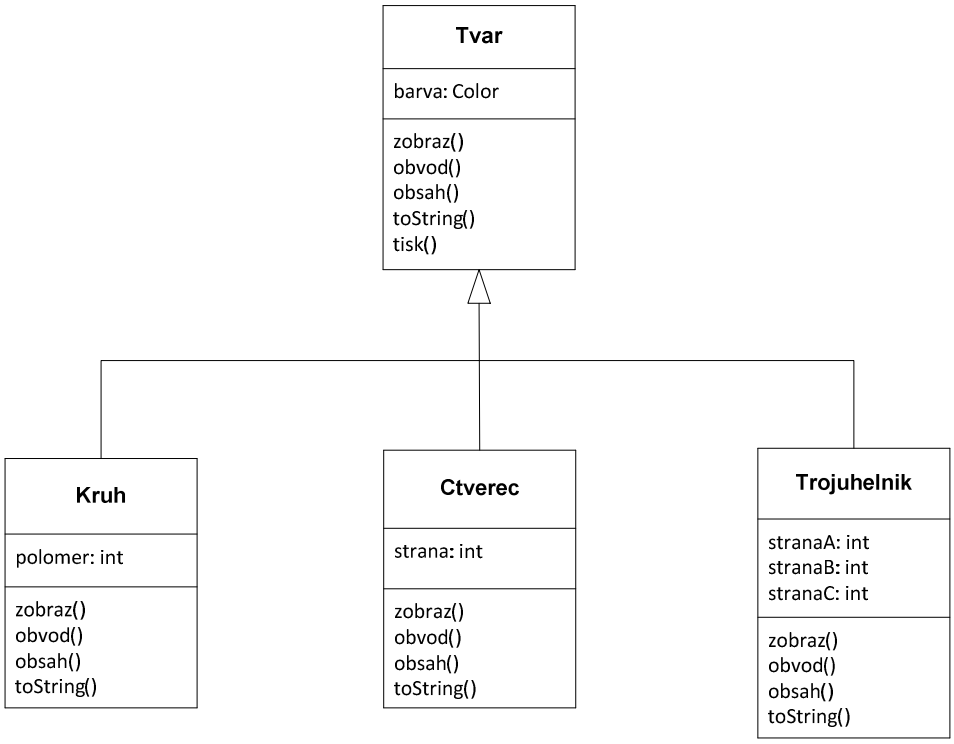
Po jejím prostudování budete schopni:

- rozumět pojmu polymorfismus,
- využívat možností abstraktní třídy,
- využívat možností rozhraní.

Klíčová slova této kapitoly:

Polymorfismus, abstraktní třída, rozhraní

V předchozích kapitolách jsme se zabývali pouze konkrétními třídami. V této kapitole si ukážeme výhody a využití abstraktních tříd, podíváme se, jak se deklaruje a využívá rozhraní a vysvětlíme si pojem a praktické využití polymorfismu (vícetvarosti) proměnné, která ukazuje na daný objekt.
V prostředí jazyka Java jsou všechny proměnné objektových typů (kromě proměnných obsahující primitivní datové typy) dynamické proměnné. To znamená, že vždy odkazují na konkrétní instanci (objekt), ale neobsahují ho. Obsahovat objekt umí tzv. statické proměnné (pozor neplést s modifikátorem static), které např. existují v jazyce C++. Z toho důvodu je prakticky v Javě pouze pozdní vazba. Vazbou se myslí vztah mezi příjemcem zaslané zprávy a metodou, kterou zaslaná zpráva vyvolá. To znamená, že až za běhu programu se přesně určí, jaká metoda se vybere k zaslané zprávě. Detailněji probereme u polymorfismu. Na jednoduchém příkladu si vysvětlíme nejdříve dědičnost a následně abstraktní třídu a rozhraní. Strukturu programu zachycuje diagram tříd UML.



Obr. 10.1 Diagram tříd řešeného příkladu

Z diagramu vidíme, že nejvyšší třída *Tvar* deklaruje datový atribut barva, metody *zobraz(), obvod()* a *obsah(),* které její podtřídy předeklarují (zastíní). Další metoda *toString()* je v podtřídách zděděna. Naopak metoda *tisk()* je deklarovaná pouze ve třídě *Tvar* a podtřídy ji vyvolávají.
Když se podíváte na kód metody *tisk()* objevíte, že tiskne dvě části:

- `this.getClass().getName()`
  pseudoproměnná *this* v tomto případě odkazuje na instanci třídy, která metodu vyvolala, což může být instance tříd *Tvar*, *Kruh*, *Ctverec*, *Trojuhelnik*. To je důležité si uvědomit. Metoda *getClass().getName()* nám vrátí jméno třídy včetně balíčků.
- `this.toString()`
  pseudoproměnná *this* odkazuje jako v předchozím případě na instanci třídy, která metodu vyvolala. Může být instance tříd *Tvar*, *Kruh*, *Ctverec*, *Trojuhelnik*.

Na první pohled se zdá, že třída *Tvar* nemá velký význam a že je skoro zbytečná. datový atribut *barva* by se dal deklarovat přímo v podtřídách.

Význam společné nadtřídy je ale v tom, že pomáhá deklarovat společné datové atributy a metody, které se mohou využívat přímo v podtřídách, jako např. metoda *toString(),* anebo se kompletně v podtřídách předeklarují, jako je to v případě metod *zobraz(), obvod() a obsah().*
Začneme kódem jednotlivých tříd a skončíme s jednoduchou aplikací. Nic nebudeme vykreslovat, u trojúhelníku předpokládáme, že se jedná o pravoúhlý trojúhelník s přeponou *stranaC*.

10.1 Třída Tvar a její podtřídy

```
import java.awt.Color;
public class Tvar {
   private Color barva;
   public Tvar (Color barva) {
      this.barva = barva;
   }

   public Color getBarva() {
      return barva;
   }

   public String toString() {
      return String.format(" barva: %s", getBarva());
   }

   public void tisk() {
      System.out.println(this.getClass().getName() +
                     this.toString());
   }

   public void zobraz() {}

   public double obsah() {
       return 0;
   }

   public double obvod() {
     return 0;
   }
}
```

Jak již bylo řečeno, smyslem deklarace třídy *Tvar* není tvorba instancí (to nemá praktický smysl), ale popis nějakých společných vlastností (metod), které budou využity v každé podtřídě.

Třída Kruh

```
import java.awt.Color;
public class Kruh extends Tvar{
   private int polomer;

   public Kruh(Color barva, int p) {
      super(barva);
      polomer = p;
   }

   public int getPolomer() {
      return polomer;
   }
   public String toString() {
      return String.format("%s\npolomer: %d",
                   super.toString(), getPolomer());
   }
```

```
    public void zobraz() {
        System.out.print("Kruh.zobraz()");
    }

    public double obsah() {
       return Math.PI * Math.pow(polomer,2);
    }

    public double obvod() {
       return 2 * Math.PI * polomer;
    }
}
```

Třída Čtverec

```
import java.awt.Color;
public class Ctverec extends Tvar{
    private int strana;

    public Ctverec(Color barva, int s) {
        super(barva);
        strana = s;
    }

    public int getStrana() {
        return strana;
    }

    public String toString() {
        return String.format("%s\nstrana: %d",
                      super.toString(), getStrana());
    }

    public void zobraz() {
       System.out.print("Ctverec.zobraz()");
    }

    public double obvod() {
        return strana * 4;
    }

    public double obsah(){
        return Math.pow(strana, 2);
    }
}
```

Třída Trojuhelnik

```
import java.awt.Color;
public class Trojuhelnik extends Tvar {
    private int stranaA, stranaB, stranaC;

    public Trojuhelnik(Color barva, int a, int b, int c) {
        super(barva);
        stranaA = a; stranaB = b; stranaC = c;
    }
```

```
   public int getA() {
      return stranaA;
   }

   public int getB() {
      return stranaB;
   }

   public int getC() {
      return stranaC;
   }

   public String toString() {
     return String.format("%s\nstranaA: %d stranaB: %d “ +
          “ stranaC: %d",
                    super.toString(), getA(), getB(), getC());
   }

   public void zobraz() {
       System.out.print("Trojuhelnik.zobraz()");
   }

   public double obvod() {
      return stranaA + stranaB + stranaC;
   }

   public double obsah(){
        return stranaA * stranaB / 2;
   }
}
```

Dále si deklarujeme třídu *GeneratorTvar*, která bude náhodně vytvářet instance od tří podtříd třídy *Tvar*.

Třída GeneratorTvaru

```
public class GeneratorTvaru {
   private Random rand = new Random();
   // pouziti metody next()
   public Tvar next() {
      switch(rand.nextInt(3)) {
         default:
         case 0: return new Kruh(Color.red, 6);
         case 1: return new Ctverec(Color.yellow, 8);
         case 2: return
                new Trojuhelnik(Color.orange, 2, 4, 7);
       }
    }
}
```

Třída *TvarTest* ukazuje využití uvedené struktury tříd. V řádcích 20 a 21 je využita metoda *tisk()*, která zobrazí název třídy (včetně jejich balíčků) a vyvolá odpovídající metodu *toString()*.

Třída TvarTest

```
1 public class TvarTest {
2   private static GeneratorTvaru gen =
3     new GeneratorTvaru();
4
5   public static void main(String[] args) {
6      Tvar[] tvar = new Tvar[9];
7      // Naplneni pole tvary:
8      for(int i = 0; i < tvar.length; i++)  {
9               tvar[i] = gen.next(); tvar[i].zobraz();
10           System.out.println(); // prazdny radek
11       }

12    // Polymorficke volani metod:
13    System.out.println("Tisk z pole");
14    for(int i = 0; i < tvar.length; i++) {
15       tvar[i].zobraz();
16       System.out.printf(" %s %.2f %s %.2f\n",
                "obsah:", tvar[i].obsah(),
17                   "obvod:", tvar[i].obvod());
18      }

19       System.out.println("Tisk pole tvar");
20       for(int i = 0; i < tvar.length; i++)
21          tvar[i].tisk();
     }
}
```

10.2 Polymorfismus (Vícetvarost)

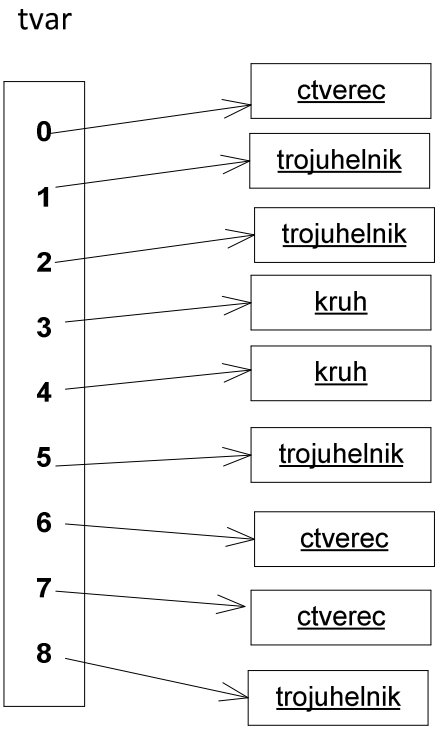
Polymorfismus znamená mnohotvárnost a je třetí elementární funkční vlastností každého objektově orientovaného programovacího jazyka. Předchozí dvě jsou *datová abstrakce* a *dědičnost*. Polymorfní operace jsou operace s mnoha implementacemi. Polymorfismus úzce souvisí s tzv. pozdní vazbou, tedy s vazbou, která je v Javě implicitní.

Brzká resp. pozdní vazba je vazba mezi objektem a vyvolanou metodou. Polymorfismus si názorně vysvětlíme na kódu metody *TvarTest*. V řádku 6 deklarujeme pole *tvar*, které bude mít reference (odkazy) na objekty (instance) třídy *Tvar*. Pole je dimenzováno na kapacitu 9 prvků. Pokud by neexistoval polymorfismus, znamenalo by to, že skutečně do pole tvar bude odkazovat na instance třídy *Tvar*. Ale když se podíváme na strukturu třídy *Tvar*, musíme konstatovat, že

vytvořit instanci od této třídy nemá praktický smysl. V předchozí kapitole o dědičnosti jsme se ale dověděli, že proměnná, která je deklarovaná (kvalifikovaná) jako proměnná nadtřídy, tak tato proměnná může za běhu programu odkazovat na instance (objekty) té stejné třídy a instance všech podtříd té konkrétní třídy. V našem případě to znamená, že proměnná kvalifikovaná jako proměnná třídy Tvar může za běhu programu odkazovat na instance tříd *Kruh*, *Ctverec* a *Trojuhelnik* (ukazovat na instanci třídy Tvar nemá smysl, jak jsme již naznačili).

V řádku 9 jsou v cyklu do pole tvar přidávány instance tříd *Kruh*, *Ctverec* a *Trojuhelnik* v náhodném pořadí. Proměnná *tvar[i]* je kvalifikovaná (deklarovaná) jako proměnná typu *Tvar*! Tuto možnost umožňuje také polymorfismus.
V řádcích 15, 16 a 17 máme další příklady polymorfismu a sice aplikace metody zobraz() na instance polymorfního (vícetvarého) pole tvar a aplikace metod *obvod()* a *obsah()* na instance polymorfního pole tvar.

To že říkáme, že pole je polymorfní v podstatě znamená, že pole obsahuje různé instance, pro které ale platí, že pole je kvalifikováno na společnou nadtřídu instancí, na které jednotlivé indexy pole odkazují.



Obr. 10.2 Schematická struktura pole *tvar* a jeho prvků (objektový diagram).

Pole *tvar* je zobrazeno na obr. 10.2. Z něho je vidět, že pole obsahuje pouze instance podtříd třídy *Tvar* a že jednotlivé indexy pole např.

tvar[4] odkazuje na instanci třídy *Kruh*. V objektovém diagramu jsou uvedeny objekty, jejichž název je podtržen.

Když pak proměnné kvalifikované na třídu *Tvar* zašleme zprávu *zobraz* nebo *obsah* nebo *obvod*, proměnná podle toho na jakou instanci konkrétní třídy ukazuje, vybere odpovídající metodu. Ukazuje-li proměnná deklarovaná jako proměnná třídy *Tvar* na instanci třídy *Kruh*, vybere se metoda *zobraz()* nebo *obvod()* nebo *obsah()* ze třídy *Kruh*! Odkazuje-li se daná proměnná na instanci třídy *Trojuhelnik*, vybere požadované metody ze třídy *Trojuhelnik*.

Obecně pak o polymorfismu můžeme říci, že je to mechanismus, podle kterého se na základě typu příjemce zprávy (jedná se o proměnnou) vybere metoda odpovídající třídy.

10.3 Abstraktní třída

Konkrétní třídy se skládaly z deklarace datových atributů a deklarace metod. Abstraktní třída je taková třída, která deklaruje alespoň jednu abstraktní metodu. Abstraktní metoda je metoda, která má „hlavičku“ (deklarovaný název metody a typy vstupních a výstupních argumentů) a nemá tělo. Tělo metody představuje kód metody deklarovaný ve složených závorkách.

```
public navratovyTyp nazev (Typ1 typ1, Typ2 typ2);
```

Příklad deklarace abstraktní metody, který vrací navratový typ, má jméno název a má dva vstupní argumenty *typ1* a *typ2*.

Abstraktní třída *Tvar* má jeden datový atribut barva, konkrétní přístupovou metodu *getBarva()* a konkrétní metody *toString()* a *tisk()*. Má tři abstraktní metody, a to *zobraz()*, *obsah()* a *obvod()*. U abstraktních metod si všimněte, že se nedeklaruje tělo metody, tedy nesmí být uvedeny ani složené závorky (jak vidíte jsou v komentáři).

Abstraktní třída Tvar

```
import java.awt.Color;
public abstract class Tvar {
   private Color barva;

   // konstruktor
   public Tvar(Color barva) {
       this.barva = barva;
   }

   public Color getBarva() {
      return barva;
   }
   public abstract void zobraz(); //{}  abstraktni metoda

   public abstract double obsah(); //{} nespecifikuje telo
```

```
   public abstract double obvod(); //{}

   // konkretni metody
   public String toString(){
        return String.format( " barva: %s", getBarva());
   }

   public void tisk(){
        System.out.println(this.getClass().getName()+
          this.toString());
  }
}
```

Třída Kruh

```
import java.awt.Color;
public class Kruh extends Tvar{
   private int polomer;

   public Kruh(Color barva, int p) {
       super(barva); polomer = p;
       }

   public int getPolomer() {
      return polomer;
   }

   public String toString() {
       return String.format("%s\n polomer: %d",
              super.toString(), getPolomer());
   }
   public void zobraz() {
      System.out.print("Kruh.zobraz()");
   }

   public double obsah() {
       return Math.PI * Math.pow(polomer,2);
   }

   public double obvod() {
        return 2 * Math.PI * polomer;
   }
}
```

Třída TvarTest

```
public class TvarTest {
   private static GeneratorTvaru gen =
       new GeneratorTvaru();

    public static void main(String[] args) {
      Tvar[] s = new Tvar[9];

      // Naplneni pole tvary:
```

```
        for(int i = 0; i < s.length; i++)      {
           s[i] = gen.next(); s[i].zobraz();
           System.out.println();
        }

        // Polymorficke volani metod:
        System.out.println("Print from array");
        for(int i = 0; i < s.length; i++) {
           s[i].zobraz();
           System.out.printf(" %s %.2f %s %.2f\n", "obsah:",
                s[i].obsah(), "obvod:", s[i].obvod());
           }

           System.out.println("----------------------------");
           for(int i = 0; i < s.length; i++) {
              s[i].tisk();
           }
    }
}
```

Třída *TvarTest* funguje stejně jako v předchozí části. Drobný rozdíl je jen v tom, že pole *tvar* je deklarované na prvku abstraktní třídy *Tvar*. Polymorfismus funguje stejně.

10.4 Rozhraní

Rozhraní na rozdíl od abstraktní třídy je svým způsobem ještě volnější. Rozhraní v Javě dovoluje deklarovat pouze hlavičky metod, tedy pouze abstraktní metody a navíc třídní (statické) neměnné (modifikátor final) datové atributy. Rozhraní na rozdíl od konkrétních a abstraktních tříd se nedědí (v podtřídách se nepoužívá klíčové slovo **extends**) ale rozhraní se implementuje (používá se klíčové slovo **implements**).

Implementace znamená, že ve třídě, která implementuje rozhraní, musí být deklarované všechny metody, které jsou uvedeny v rozhraní. V našem případě to znamená, že každá třída musí deklarovat metody *zobraz()*, *obsah()*, *obvod()* a *tisk()*. Záleží ale na každé třídě, jak to provede. Třídy implementující rozhraní musí dodržet hlavičky metod, vlastní tělo metod si určují sami.

Rozhraní Tvar

```
public interface Tvar {
   static final int cislo = 27;
   public void zobraz(); //{}  implicitni modifikator public
   public double obsah(); //{}
   public double obvod(); //{}
   public void tisk();
}
```

Třída Ctverec

```
import java.awt.Color;
public class Ctverec implements Tvar {
   private Color barva;
   private int strana;

   // konstruktor
   public Ctverec(Color barva, int s) {
      this.barva = barva; strana = s;
   }

   public void zobraz() {
      System.out.print("Ctverec.zobraz()");
   }

   public double obvod() {
      return strana * 4;
   }

   public double obsah(){
      return Math.pow(strana, 2);
   }


   public int getStrana(){
      return strana;
   }

   public Color getBarva() {
      return barva;
   }

   public String toString(){
      return String.format(" barva: %s strana: %d",
                  getBarva(), getStrana());
   }

   public void tisk() {
      System.out.println(this.getClass().getName() +
                      this.toString());
   }
}
```

Třída TvarTest

```
public class TvarTest {
   private static GeneratorTvaru gen =
          new GeneratorTvaru();

   public static void main(String[] args) {
      Tvar[] tvar = new Tvar[9];

      // Naplneni pole tvary:
      for(int i = 0; i < tvar.length; i++) {
       tvar[i] = gen.next(); tvar[i].zobraz();
```

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

         // Polymorficke volani metod:
         System.out.println("Print from array");
       for(int i = 0; i < tvar.length; i++) {
           tvar[i].zobraz();
           System.out.printf(" %s %.2f %s %.2f\n", "obsah:",
                   tvar[i].obsah(), "obvod:", tvar[i].obvod());
       }

          System.out.println("----------------------------");
       for(int i = 0; i < s.length; i++) {
          tvar[i].tisk();
       }
  }
}
```

Třída *TvarTest* je stejná jako v předchozích případech jen s tím rozdílem, že pole tvar je kvalifikováno jako rozhraní.

10.5 Rozhraní – další příklad

V souvislosti se zapouzdřením se seznámíme se dvěma termíny:
Rozhraní třídy je obecně množina informací, které o sobě třída zveřejní. Mezi rozhraní patří např. vše, co je ve třídě označeno modifikátorem public. Rozhraní – anglicky interface.

Implementace je vše, co má být před uživatelem ukryto. Je to způsob, jak je třída naprogramována, jak dělá to co umí.

Java zavedla typ rozhraní explicitně. V rozhraní jsou deklarovány pouze metody. Rozhraní totiž na rozdíl od tříd neříká vůbec nic o tom, jak budou jednotlivé metody implementovány. Pro snazší pochopení si uvedeme jednoduchý příklad směnárny, která bude realizována jako směnárna bez poplatku a směnárna s poplatkem. Ve výpisu bude nejdříve uvedeno rozhraní, dále pak třída směnárna bez poplatku, směnárna s poplatkem a třída směnárna test.

Rozhraní Smenarna

```
public interface Smenarna {
   double vykup(double kolik);
   double prodej(double kolik);
}
```

Třída SmenarnaBezPoplatku

```
public class SmenarnaBezPoplatku implements Smenarna {
```

```
    private double kurz;

    // konstruktor
    public SmenarnaBezPoplatku(double kurz) {
          this.kurz = kurz;
    }

    public double getKurz() {
       return kurz;
    }

    public double vykup(double kolik) {
       return kolik / kurz;
    }

    public double prodej(double kolik) {
       return kolik * kurz;
    }
}
```

Třída SmenarnaSPoplatkem

```
public class SmenarnaSPoplatkem implements Smenarna {
    private double kurz;
    private double pMin;
    private double pProc;

    public SmenarnaSPoplatkem(double kurz, double pMin,
                                        double pProc) {
       this.kurz = kurz; this.pMin= pMin; this.pProc = pProc;
    }

    public double getKurz(){
        return kurz;
    }

    public double getpMin() {
       return pMin;
    }

    public double getpProc() {
       return pProc;
    }

    public double vykup(double kolik) {
       double castkaBezPoplatku = kolik / getKurz();
        double poplatek = castkaBezPoplatku * pProc / 100;
        if(poplatek < pMin) poplatek = pMin;
        return castkaBezPoplatku - poplatek;
    }

    public double prodej(double kolik) {
```

```
        double castkaBezPoplatku = kolik * getKurz();
        double poplatek = castkaBezPoplatku * pProc / 100;
        if( poplatek < pMin) poplatek = pMin;
        return castkaBezPoplatku - poplatek;
    }
}
```

Pro úplnost ještě uvádíme třídu *SmenarnaTest*,která ukazuje na možné použití resp. blíže objasní uvedený příklad.

Třída SmenarnaTest

```
public class SmenarnaTest {
   public static void main(String[] args) {
       SmenarnaBezPoplatku sbp =
            new SmenarnaBezPoplatku(1/28.3);

       SmenarnaSPoplatkem ssp =
             new SmenarnaSPoplatkem(1/28.3, 20, 7);


       double castkaKc = 3000;
       double castkaEu = 300;
       double v1 = sbp.vykup(castkaEu);

       System.out.printf(
           "v1 vykup castka: Eu %7.2f castka Kc: %7.2f\n"
                    ,castkaEu, v1);

       double v2 = sbp.prodej(castkaKc);
       System.out.printf(
             "v2 prodej castka: Kc %7.2f castka Eu: %7.2f\n"
                    ,castkaKc, v2);

       double v3 = ssp.vykup(castkaEu);
       System.out.printf(
              "v3 vykup castka: Eu %7.2f castka Kc: %7.2f\n"
                    ,castkaEu,v3);

       double v4 = ssp.prodej(castkaKc);
       System.out.printf(
              "v4 prodej castka: Kc %7.2f castka Eu: %7.2f\n"
                    ,castkaKc, v4);
   }
}
```

Deklarace rozhraní obsahuje deklarace metod *výkup*() a *prodej*(), obě s jedním argumentem. Obě metody jsou implementovány různě ve třídách, které implementují rozhraní. Třída *SmenarnaBezPoplatku* implementuje obě metody odlišně než třída *SmenarnaSPoplatkem*.

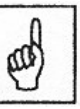
Formálně ale obě třídy musí dodržovat deklaraci uvedenou v rozhraní pro obě metody.

Rozhraní dále tvoří souhrn informací, které se z hlaviček deklarovaných metod vyčíst nedají. Jedná se o podmínky, za kterých daná metoda pracuje, informace o tom, že daná instance je jedináček atd. Tento souhrn informací bývá označován jako kontrakt (smlouva) a měl by být uveden v dokumentačních komentářích třídy, jejich metod a atributů.

Součástí dokumentace (a tím i obecně závazného kontraktu) by měly být následující informace:

- popis funkce metody,
- význam jednotlivých parametrů a omezení na ně kladená,
- další podmínky, které je třeba při volání metody splnit,
- význam funkční hodnoty,
- popis chybových situací k nimž může dojít.

Rozhraní jako takové nemůže samo o sobě vytvořit instanci. Proto pokud hovoříme o instanci rozhraní, máme tím na mysli instanci některé z tříd, která dané rozhraní implementovala.



Rozhraní je v podstatě další **zobecnění abstraktní třídy**. **Rozdíly mezi rozhraním a abstraktní třídou jsou následující:**

- Z formálního hlediska se místo klíčového slova abstract používá klíčové slovo interface.
- Třída může implementovat libovolný počet rozhraní, ale může být podtřídou pouze jedné abstraktní třídy (nebo jedné konkrétní, samozřejmě).
- Abstraktní třída může mít i neabstraktní metody, rozhraní má pouze „abstraktní“ metody.
- Abstraktní třída může deklarovat datové atributy, rozhraní může deklarovat pouze konstanty typu static final (třídní a neměnné).
- Abstraktní třída může pro své metody použít modifikátory private, protected, public, nebo nic – přístup pouze v rámci balíčku, všechny metody rozhraní jsou implicitně public (i když nemají žádný modifikátor).
- Abstraktní třída může definovat konstruktory, rozhraní ne.

To že třída může implementovat libovolný počet rozhraní, umožňuje řešit problémy vícenásobné dědičnosti. Jak jste si všimli, Java umožňuje pouze jednoduchou dědičnost, každá třída má pouze jednu nadtřídu.

Hlavní důvod pro zavedení rozhraní je schopnost zobecnit objekt na více možných typů. Druhý důvod je stejný jako u rodičovských abstraktních tříd – jednak bráníme programátorovi ve vytvoření objektu této třídy, a jednak poskytujeme informaci, že tato třída slouží jako rozhraní. Je-li možné vytvořit konkrétní základní třídu bez jakékoli definice metod, nebo datových atributů, je dobré dát přednost rozhraní před abstraktní třídou. Máte-li představu co bude konkrétní, základní třídou, měli byste se nejdříve pokusit o vytvoření rozhraní. K volbě abstraktní třídy byste se měli uchýlit teprve, až budete moci rozšířit rozhraní o definici metod, nebo datových atributů (členských proměnných). Poslední možností je vytvoření abstraktní třídy.

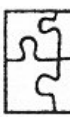
Abstraktní třída a rozhraní jsou dva důležité pojmy v objektově orientovaném programování.
Rozhraní je v podstatě „čistá“ abstraktní třída obsahující pouze „hlavičky“ metod.

Vyjmenujte rozdíly mezi abstraktní třídou a rozhraním? Jak je rozhraní implementováno?

Polymorfismus patří k základním charakteristikám objektově orientovaného přístupu. V této souvislosti je třeba také pochopit význam tzv. polymorfního (vícetvarého) objektu. Je to objekt kvalifikovaný v nadtřídě, který ukazuje na objekty svých podtříd.

10.1 Může být rozhraní organizováno v hierarchii?

10.2 Proč se objekt kvalifikuje většinou na typ rozhraní?



10.1 Ano, rozhraní mohou být organizována v hierarchii.

10.2 Tím že je objekt kvalifikován jako rozhraní mu umožňuje, aby mohl ukazovat na všechny objekty, které to rozhraní implementují.

V této kapitole jsme si ukázali na praktických příkladech, jak se dá využívat mechanismus abstraktní třídy a mechanismus rozhraní. Použití rozhraní lépe ještě vynikne při spojení s nějakým seznamem, do kterého můžeme například ukládat všechny konkrétní objekty, které implementují dané rozhraní.

11 Využití polí pro ukládání objektů

V této kapitole se dozvíte:

- jak musíme strukturovat třídu, abychom do ní mohli ukládat další objekty,
- jaké jsou základní metody, které tato třída potřebuje,
- jak je možné z takové třídy vytvořit „kruhovou“ frontu.

Po jejím prostudování budete schopni:

- využívat pole pro vytvářejí jednoduchých statických seznamů,
- vytváření zásobníků a front.

Klíčová slova této kapitoly:

třída registr, implementace fronty

Pole jako taková se probírají v algoritmech a datových strukturách. Proto se v této kapitole soustředíme na to, jak využít strukturu pole v jednoduchém seznamu, kterému můžeme říkat registr.

Hlavní nevýhodou při standardní práci s poli je, že většinou pracujeme přímo s polem a nevyužíváme metody, které bychom si mohli deklarovat, pokud by uvedené pole sloužilo jako datový atribut již zmíněné struktury registr. Navíc v příkladech pro práci s poli se implicitně předpokládá, že všechny prvky pole budou obsazeny a takto vždy pracujeme jen s funkcí *length*, která vrací kapacitní velikost pole. Ta může být v mnoha směrech podobná zásobníku (přidáváme prvky na konec seznamu a odebíráme prvky z konce seznamu), ale navíc můžeme zpřístupnit prvek podle indexu, přidat základní metodu hledání, tisku atd.

Základní operace (metody), které bychom od takto deklarované třídy požadovali by byli následující:

- vložení prvku (na konec seznamu),
- odebrání prvku (z konce seznamu),
- vytvoření textové reprezentace všech prvků pole,
- vytištění textové reprezentace všech prvků pole,
- vrácení prvku pole na pozici zadané indexem,
- kapacita pole,
- obsazenost pole,

- zda je seznam prázdný,
- zda je seznam plný atd.

Nejdříve si ukážeme strukturu číselného seznamu, pak přejdeme k seznamu, do kterého se dají ukládat odkazy na obecné objekty.

11.1 Třída Registr pro primitivní typy

```
public class Registr {
   private int pole[];
   private int top;

   // konstruktor
   public Registr(int pocet) {
      pole = new int[pocet];
      top = -1;
   }

   public int getTop() {
       return top;
   }

   public void incrTop() {
       top++;
   }

   public void decrTop() {
       top--;
   }

   public void vlozit(int prvek) {
       if ((getTop() + 1) < pole.length)
        { incrTop(); pole[top] = prvek; }
          else System.out.println("Registr je obsazeny");
   }

   public String toString() {
      String t = String.format("%5s %9s\n","Index","Hodnota");
      for (int i=0; i<=top; ++i)
          t = t+ String.format("%5d %9d\n",i, pole[i]);
      return t;
    }

    public void tisk() {
         System.out.println(this.toString());
    }

    public int getPrvek(int i) {
       int prvek=0;
       if (i>=0 && i<= top ) // top ukazuje skutecnou velikost
         prvek = pole[i];
       else System.out.printf("%s %d %s\n","Index",i,
                                          "mimo rozsah");
       return prvek;
    }
```

```
    public int vyhledat(int prvek) {
       int vysl = -1;
       for (int i=0; i<= top; i++)
         if (pole[i]== prvek) {
              vysl = i; break;
              }
       return vysl;
    }

    public void odstranit(int prvek) {
       int v = vyhledat(prvek);
       if (v > -1 ) {
         for (int i=v; i< top; i++)
            pole[i] = pole[i+1];
            decrTop();
         System.out.println("Prvek odstranen"); }
         else System.out.printf("%s %d %s\n",
                 "Prvek",prvek,"neni v registru");
         }


    public int minimum() {
       int min = pole[0];
       for (int cislo: pole)
           if (cislo < min) min = cislo;
        return min;
    }
```

```
1   public Registr novyReg(int limit) {
2      Registr novy = new Registr(getTop());
3      for(int i = 0; i <= getTop(); i++)
4      if(getPrvek(i) < limit)
5               novy.vlozit(getPrvek(i));
6      return novy;
    }
    public void vymazatVse() {
       top = -1;
    }
}
```

Zajímavé je deklarace a použití metody nazvané *novyReg()*. Tato metoda má za úkol vytvořit a vrátit odkaz na nový registr, do kterého se zkopírují všechny prvky původního registru, které vyhovují podmínce, v našem případě jsou menší než argument limit, který předáváme do metody. Návratový typ tedy musí být *Registr*. V řádku 2 metody deklarujeme proměnnou novy, která bude ukazovat na instanci třídy *Registr* a bude mít velikost rovnou počtu prvků aktuálního registru (parametr *getTop()*).

Budeme pak procházet daným registrem a všechny prvky, které vyhovují nastavené podmínce vložíme do nového registru. Odkaz na tento nový registr vrátíme v řádku 6 – příkaz return.

Třída RegistrTest

```
1 public class RegistrTest {
2   public static void main(String[] args) {
3      Registr a = new Registr(5);
4      a.vlozit(10);
5      a.vlozit(15);
6      a.vlozit(-44);
7      a.tisk();
8      a.odstranit(2);
9      a.tisk();
10     System.out.println("Minimalni prvek registru: " +
                                  a.minimum());
11        int bb= a.getPrvek(15);
12     System.out.println("Nalezeny prvek : " + bb);
13     Registr b = new Registr(5);
14     b.vlozit(15); b.vlozit(25); b.vlozit(33);
15     Registr c = b.novyReg(20);
16     c.tisk();
   }
}
```

Třída *RegistrTest* ukazuje na možnosti použití instancí třídy *Registr*. V řádku 15 deklarujeme proměnnou c jako typ Registr, který bude obsahovat všechny prvky registru, na který odkazuje proměnná b, které jsou menší než 20.

Pro třídy *Registr* platí:

- pole je vnitřní datový atribut třídy *Registr*,
- top – ukazatel na místo, kam se má vložit nový prvek,
- třída *Registr* obsahuje všechny potřebné metody, další nutné metody můžeme doplnit,
- velikost datového atributu pole se určí až za běhu programu prostřednictvím konstruktoru,
- pozor na problém indexů:
    - číslování 0 – (n-1) můžeme ponechat,
    - anebo zvolit indexování 1 - n

11.2 Třída Registr pro objektové typy

Nyní si ukážeme, jak se změní kód třídy *Registr*, když do něho budeme ukládat např. objekty třídy *Osoba*. Uvedeme proto nejdříve zredukovaný kód třídy *Osoba* a pak upravenou třídu *Registr*.

Část třídy Osoba

```
public class Osoba {
   private String jmeno;
   private int rokNarozeni;
   private int vaha;

   // deklarace konstruktoru
   public Osoba() {
     this("neuvedeno", 0, 0);
   }

   public Osoba(String jmeno, int rokNarozeni, int vaha) {
     this.jmeno= jmeno; this.rokNarozeni = rokNarozeni;
        this.vaha = vaha;
   }

   public String toString() {
      return String.format("%5s %s %15s %4d %5s %d\n",
          "Jmeno",getJmeno(),"rok narozeni:",getRokNarozeni(),
          "vaha:",getVaha());
   }

   public void tisk() {
       System.out.println(this.toString());
   }
}
```

Třída RegistrOsob

```
public class RegistrOsob {
   private Osoba pole[];
   private int top;

   // konstruktor
   public RegistrOsob(int pocet){
      top = -1;
      pole = new Osoba[pocet];
   }

   public void vlozit(Osoba prvek) {
      if ((top + 1) < pole.length)
           { top += 1; pole[top] = prvek; }
      else System.out.println("Registr je obsazeny");
   }

   public Osoba getPrvek(int i) {
      Osoba prvek=null;
       if (i>=0 && i< pole.length)
        prvek = pole[i];
      else System.out.printf(
           "%s %d %s\n","Index",i,"mimo rozsah");
    return prvek;
   }

   public int vyhledat(String jmeno) {
      int vysl = -1;
```

```
   for (int i=0; i<= top; i++) {
     System.out.println(" i, jmeno: "+i+"
              "+pole[i].getJmeno());
        if (pole[i].getJmeno()== jmeno)
           {vysl = i; break; }}
     return vysl;
    }

public int vyhledat(Osoba prvek) {
    int vysl = -1;
       for (int i=0; i<= top; i++)
       { System.out.println("Index: "+i);
         if (pole[i]== prvek)
          { vysl = i; break; }}
       return vysl;
    }

public void odstranit(Osoba prvek) {
  int v = vyhledat(prvek);
  if (v == -1) System.out.printf("%s %d %s\n",
                "Prvek",prvek,"neni v registru");
       else this.odstranit(v);
    }

public void odstranit(String jmeno) {
     int v = vyhledat(jmeno);
      if (v == -1) System.out.printf("%s %s %s","Jmeno:",
                jmeno,"neni v registru");
        else this.odstranit(v);
    }

    public void odstranit(int v) {
      for (int i=0; i< top - v; i++)
          pole[v+i] = pole[v+i+1];
          top -= 1;
          System.out.println("Prvek odstranen");
    }

public String toString() {
      String t = String.format("%5s %16s\n",
                              "Index","Hodnota");
        for (int i=0; i<=top; ++i)
          t = t+ String.format("%5d ",i)+pole[i].toString();
      return t;
}

public void tisk() {
   System.out.println(this.toString());
   }
```

```
    // pozor na pouziti zjednoduseneho for
    public Osoba minVaha() {
        Osoba min = pole[0];
        for (Osoba zaznam: pole) {
          if (zaznam != null)
           if (zaznam.getVaha() < min.getVaha()) min = zaznam;
      }
        return min;
     }

    public Osoba maxVek() {
      Osoba max = pole[0];
         for (int i=0; i<= top; i++)
           if (pole[i].getVek(2005)> max.getVek(2005))
              max = pole[i];
          return max;
      }
}
```

Hlavní změna ve třídě *RegistrOsob* je v tom, že si musíme důsledně pamatovat, že pracujeme s objektovými. To se např. projeví v metodách *minVaha()* a *maxVek()*. Tak musíme pamatovat na to, že na konkrétní prvek pole ještě musíme použít přístupovou metodu, abychom se dostali ke konkrétní uložené hodnotě.

Uvnitř metody *minVaha*() používáme zkrácený cyklus for, který má pro nás tu nepříjemnou vlastnost, že vždy prochází celým polem (až po *pole.length*). My však celé pole nemusíme mít obsazené, od toho máme datový atribut top, který ukazuje na poslední obsazenou pozici v poli. Proto musíme otestovat, zda načtený prvek se nerovná null. Pokud bychom to neošetřili, vyvolalo by to výjimku a konec programu způsobenou tím, že příjemce zprávy je null a ten nereaguje na námi deklarované zprávy.

Třída *Registr* má svou funkčností velmi blízko k datové struktuře zásobník. Proto je pro nás jistě zajímavější se podívat, jak by se dalo využít pole pro datovou strukturu fronta. Datová struktura zásobník vystačí s jednou „režijní“ proměnnou, která ukazuje na aktuální pozici. Datová struktura fronta potřebuje zabezpečit přidávání na konec a odebírání ze začátku. Abychom navíc zabezpečili obsazování celé fronty, vytváříme tzv. kruhovou frontu. To znamená, že když se dostaneme na konec pole, pokračujeme ještě v začátku pole. K tomu potřebujeme ještě další „režijní“ atribut a tím je aktuální počet prvků ve frontě. Pro vlastní operaci přechodu z konce pole na začátek se používá operace zbytek po dělení, které má symbol %. Např.
6 % 4 = 2
6 % 7 = 1
6 % 21 = 3

11.3 Třída Fronta

```
public class Fronta {
   private int a[];
   private int pocet, zapisUk, cteniUk;

   public Fronta(int n)   {
       a = new int[n];
      zapisUk = 0; pocet = 0;
      cteniUk = -1;
    }

   public void zapis(int prvek) {
     if (pocet >= a.length)
         System.out.println(
         "Fronta je plna - zapis neproveden");
      else { pocet++; a[zapisUk] = prvek;
          zapisUk = ((zapisUk+1) % a.length); }
   }

   public int cteni() {
      int prvek = 0;
       if (pocet <= 0)
        System.out.println("Fronta je prazdna");
      else { pocet--;
            cteniUk = (cteniUk + 1) % a.length;
             prvek = a[cteniUk]; }
      return prvek;
   }

   public int velikost() {
     return pocet;
     }

   public boolean prazdna() {
       return (pocet == 0);
   }

   public boolean plna() {
       return (pocet == a.length);
   }

   public int kapacita() {
       return a.length;
   }

   public String toString() {
     int index = cteniUk;
        String t="";
        t = String.format("%s\n%s%13s\n",
                  "Fronta vypis","Index","Hodnota");
        for (int i=0; i< pocet; i++)  {
         index = (index +1) % a.length;
             t += String.format("%5d%13d\n",index,+a[index]);
         }
   return t;
 }
```

```
    public void tisk() {
        System.out.println(this.toString());
    }
}
```

Metody *tisk*() a *toString*() jsou navíc pro naši kontrolu. Jak vidíte z uvedeného kódu, třída má tři pracovní datové atributy pro správu fronty a to ukazatel na zápis, ukazatel na čtení a počet uložených prvků ve frontě. V metodách čtení a zápisu jsou ukazatele modifikovány s využitím operace zbytek po celočíselném dělení. To právě umožňuje vytvořit „kruhový seznam.

Jazyk Java deklaruje rozhraní jako samostatný typ. To je významný posun oproti jiným objektově orientovaným jazykům. Praxe jednoznačně potvrdila, výhodnost tohoto samostatného typu, který umožňuje tvorbu programových komponent.

Vyjmenujte základní operace pro strukturu seznam a promyslete jejich implementaci.

Proč třída implementuje více rozhraní, co jí to umožňuje?

11.1 Jak se liší deklarace třídy *Registr* pro ukládání např. objektů třídy Osoba od ukládání např. celočíselných hodnot.

11.2 Jakým jiným způsobem by se dala realizovat datová struktura fronta. (Pokud bychom nepoužily tzv. kruhový buffer).

11.1 Pouze v deklaraci datové struktury a pak ve všech deklaracích prvků pole se musí objevit odpovídající typ. Generické typy umožňují deklaraci seznamů ještě zefektivnit viz. další semestr.

11.2 Datová struktura fronta by se dala také realizovat tak, že po odebrání prvku ze začátku fronty bychom všechny prvky posunuly o jeden směrem k nultému indexu (začátku pole) a následně pak upravili

pomocné proměnné. Podobným způsobem pracuje metoda pro odstranění prvku z registru.

Deklarace vlastního seznamu formou třídy *Registr* se může jevit zbytečná když existuje efektivní knihovna kontejner, která obsahuje všechny základní datové struktury typu seznam. Výhodou třídy *Registr* je v tom, že názorně ukazuje, jak lze objektově využít datovou strukturu pole a jak lze aplikovat základní metody. To se dá pak snadno využívat např., v metodě, která vytváří nový *registr* obsahující pouze vyprané prvky původního seznamu.

12 Spojový seznam, dědičnost a kompozice (delegace) při vytváření vlastních seznamů

V této kapitole se dozvíte:

- proč se zavádí a jakou mají funkci obalové třídy,
- jakou strukturu má spojový seznam,
- jaké třídy je třeba deklarovat pro realizaci spojového seznamu,
- jak se používá dědičnost při vytváření vlastního seznamu z již hotových tříd,
- jak se využívá skládání (delegování) pro tvorbu vlastního seznamu z již hotové třídy.

Po jejím prostudování budete schopni:

- využívat obalové třídy,
- navrhnout a vytvořit spojový seznam,
- využít dědičnosti na tvorbu vlastního seznamu,
- využít skládání (delegování) na tvorbu vlastního seznamu.

Klíčová slova této kapitoly:

spojový seznam, uzel, dědičnost, skládání, delegování

12.1 Obalové třídy (Wrapper Classes)

Dříve než se pustíme do spojových seznamů, musíme si vysvětlit co jsou to tzv. obalové třídy a k čemu slouží. Víme, že existují dvě velké skupiny typů, a to primitivní typy a objektové typy. Výhodou primitivních typů je, že daná proměnná přímo obsahuje hodnotu primitivního typu, nemá tedy žádné nároky na režijní (pomocné) struktury, které jsou nutné zavést pro objektové typy. Další velkou výhodou primitivních typů jsou implicitní knihovny pro práci s primitivními typy. Jen se vezměme například číselné hodnoty a celou spoustu aritmetických operací, které jsou součástí implicitních knihoven.

Velkou nevýhodou primitivních typů je to, že se na ně tedy nedá odkazovat referencí jako na ostatní objektové typy. To se objevuje jako problém u tříd, které jsou např. deklarované jako kolekce prvků a tyto prvky mohou být pouze objektové typy. Aby se tento problém vyřešil, byly zavedeny obalové třídy a to tak, že každý primitivní typ má odpovídající třídu obalovou třídu v balíčku java.lang.

Jsou deklarované následující obalové třídy (anglicky type-wrapper):

Boolean, Byte, Character, Double, Float, Integer, Long, Short

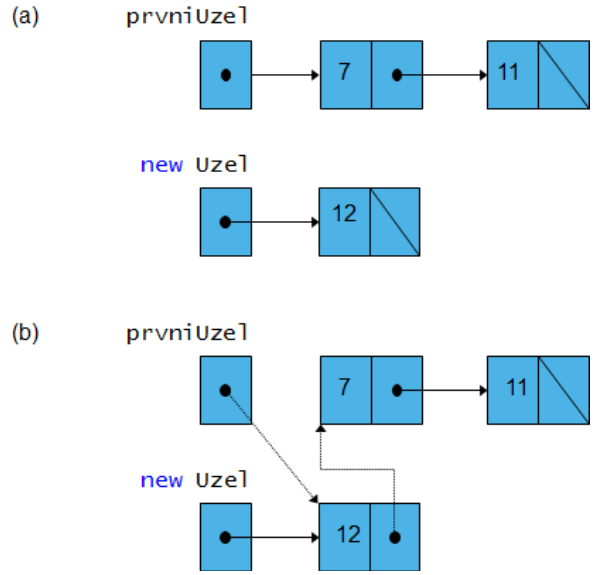
Obalové třídy nemohou manipulovat s proměnnými primitivních typů jako s objekty, ale dovolují manipulovat s objekty typových obalových tříd. Každá z obalových tříd je podtřídou třídy *Number*. Obalové třídy jsou deklarovány jako final třídy, nemohou tedy mít podtřídy.

```
int i2 = 2;
Integer ctyri = 2 * i2;
int i6 = 2 + ctyri;
```

Jak vidíte z kratičké ukázky práce s nimi je zcela intuitivní.

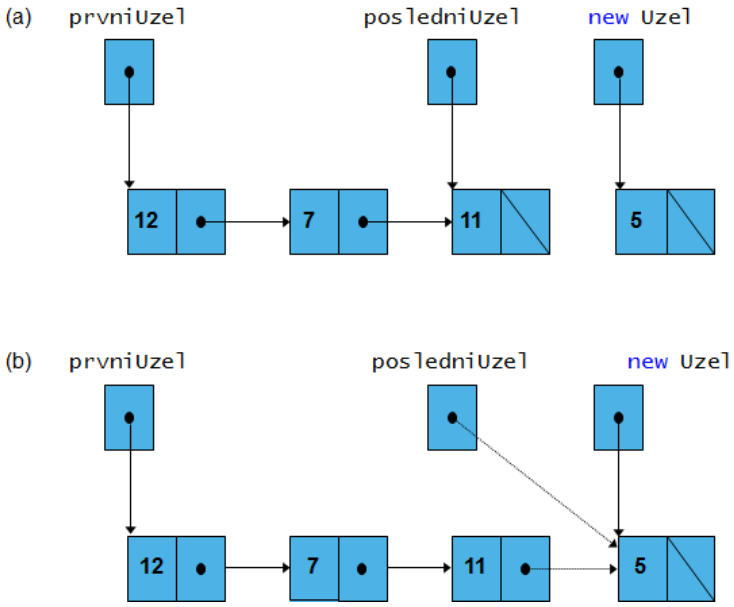
12.2 Spojový seznam

Spojový seznam je lineární datová struktura, která je charakteristická tím, že její prvky (říkáme jim uzly) odkazují sami na sebe.

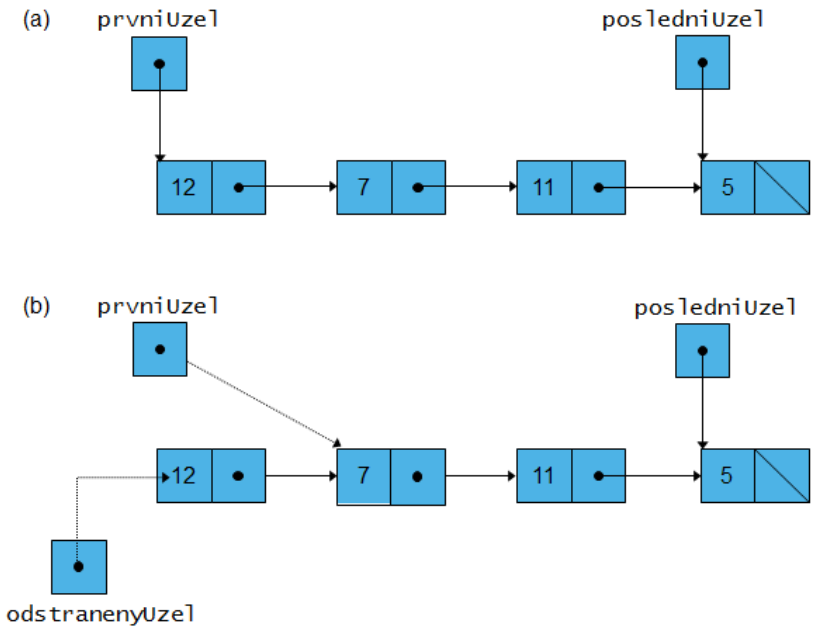


Obr. 12.1 Přidání dalšího uzlu na začátek spojového seznamu

Jak vidíte z obrázků, pracujeme pouze s jednosměrnými spojovými seznamy. Existují samozřejmě obousměrné spojové seznamy, které umožňují vkládat a odebírat uzly ze spojového seznamu nejen ze začátku a konce.



Obr. 12.2 Přidání nového uzlu na konec spojového seznamu



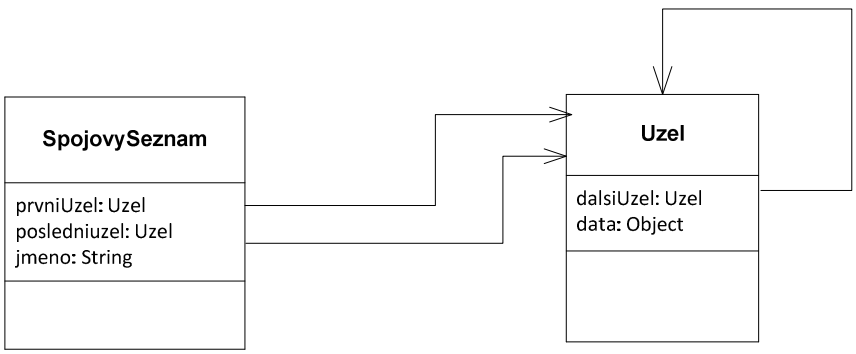
Obr. 12.3 Odstranění uzlu ze začátku spojového seznamu

Na obrázcích jsou uvedeny základní operace s jednosměrným spojovým seznamem. Pokud chceme vypracovat spojový seznam, musíme deklarovat a vytvořit odpovídající třídy. V našem případě se jedná o třídy *Uzel* a *SpojovySeznam*.

12.2.1 Třída Uzel

Třída *Uzel*, jak je patrno z předchozích obrázků deklaruje datový atribut do něhož může být buď uložena hodnota primitivního datového typu (jak je to v našem příkladu), nebo odkaz (reference) na objektový datový typ, tedy na instanci např. třídy *Osoba*. Dalším datovým atributem je odkaz na další uzel. Dále třída obsahuje konstruktor a přístupové metody.

```
public class Uzel {
   private Object data;  //datovy atribut
   private Uzel dalsiUzel; // referencni atribut

   // konstruktor vytvarejici uzel, vazba na object
   public Uzel( Object object ) {
      this( object, null );
   }

   // konstruktor tvorici uzel s vazbou na dalsi uzel
   public Uzel( Object object, Uzel node ) {
      data = object;
      dalsiUzel = node;
   }

   // pristup k datove slozce uzlu
   public Object getData()  {
    // vraci atribut data daneho uzlu
      return data;
   }

   public Uzel getDalsiUzel() {
   // dej dalsi uzel
      return dalsiUzel;
   }

   public void setDalsiUzel(Uzel nUzel) {
       dalsiUzel = nUzel;
   }
}
```



Obr. 12.4 Diagram tříd jednosměrného spojového seznamu

Třída *EmptyListException* je uvedena navíc. Výjimky (Exceptions) probírat nebudeme. Tato výjimka ošetřuje případ, kdy je spojový seznam prázdný. Bez použití výjimek bychom to řešili tak, že by se vypsala o záležitosti zpráva na konzolu.

Třída EmptyListException – výjimka prázdného seznamu

```
public class EmptyListException extends RuntimeException{
   // konstruktor bez parametru
   public EmptyListException() {
   // call other EmptyListException constructor
      this( "Spojovy seznam" );
   }

   // konstruktor s jednim parametrem
   public EmptyListException( String name ) {
      super( name + " je prazdny" ); // vola konstruktor
nadtridy
   }
}
```

12.2.2 Třída SpojovySeznam

Tato třída je vlastně tvůrcem spojového seznamu a využívá ke své činnosti dříve deklarovanou třídu *Uzel*. Jak je vidět z deklarace, má tři datové atributy. První dva ukazují na začátek resp. konec spojového seznamu a třetím datovým atributem je jméno spojového seznamu. To může být důležité, pokud bychom pracovali s více spojovými seznamy. Nejdůležitějšími metodami této třídy jsou čtyři metody:

- *insertAtFront()* – vloží uzel na začátek spojového seznamu,
- *insertAtBack()* – vloží uzel na konec spojového seznamu,
- *removeFromFront()* – odebere a tím odstraní uzel ze začátku spojového seznamu,
- *removeFromBack()* – odebere a tím odstraní uzel z konce spojového seznamu.

Spojovým seznamem můžeme procházet jednosměrně, proto je operace *removeFromBack()* obtížnější. Úpravu spojového seznamu musíme začít od prvního uzlu a pokračovat rekurzivně až na poslední uzel.

Třída Spojový seznam

```
public class SpojovySeznam {
   private Uzel prvniUzel;
   private Uzel posledniUzel;
   private String jmeno; // nazev seznamu pro tisk

   // konstruktor tvorici prazdny seznam se jmenem “list”
   public List()  {
      this( "Spojovy seznam" );
   }

   // konstruktor tvorici prazdny seznam se zadanym jmenem
   public SpojovySeznam( String jmenoSeznamu ) {
      jmeno = jmenoSeznamu;
      prvniUzel = posledniUzel = null;
   }

   // vlozi Object na zacatek seznamu
   public void insertAtFront( Object insertItem ) {
    // prvniUzel a posledniUzel odkazuji na stejny objekt
      if ( isEmpty() )
         prvniUzel = posledniUzel = new Uzel( insertItem );
      else
        // prvniUzel odkazuje na novy uzel
         prvniUzel = new Uzel( insertItem, prvniUzel );
   }

   // vlozi Object na konec seznamu (List)
   public void insertAtBack(Object vlozenaData ) {
     // prvniUzel a posledniUzel odkazuji na stejny Object
      if ( isEmpty() )
         prvniUzel = posledniUzel = new Uzel( vlozenaData );
      else {

     // posledniUzel a jeho dalsiUzel odkazuje na novy uzel
         posledniUzel = posledniUzel.getDalsiUzel();
         posledniUzel.setDalsiUzel(new Uzel(vlozenaData));
         }
   }

   // odstran prvni uzel ze seznamu
   public Object removeFromFront() throws EmptyListException {
      if ( isEmpty() ) // throw exception if List is empty
         throw new EmptyListException( jmeno );

      // zpristupneni odstranovanych dat
      Object odstranenaData = prvniUzel.getData();

      // aktualizace odkazu prvniUzel a posledniUzel
      if ( prvniUzel == posledniUzel )
         prvniUzel = posledniUzel = null;
      else
         prvniUzel = prvniUzel.getDalsiUzel();

      return odstranenaData; // vraci data odstraneneho uzlu
   }
```

```
   // odstraneni posledniho uzlu ze seznamu
   public Object removeFromBack() throws EmptyListException  {
      if ( isEmpty() ) // throw exception if List is empty
            throw new EmptyListException( jmeno );

      // zpristupni data odstraneneho uzlu
      Object removedItem = posledniUzel.getData();

      // aktualizuje odkazy na prvnihoUzel a posledniUzel
      if ( prvniUzel == posledniUzel )
           prvniUzel = posledniUzel = null;
      else  {
          // umisti novy uzel jako poslední
         Uzel aktualni = prvniUzel;

         // cykluje dokud current uzel neodkazuje na
         // posledniUzel
         // rekurze
          while ( aktualni.getDalsiUzel() != posledniUzel )
             aktualni = aktualni.getDalsiUzel();

           // aktualni je novy posledniUzel
            posledniUzel = aktualni;
            aktualni.setDalsiUzel(null);
         } // end else

       return removedItem;
   } // end method removeFromBack

   // zjisteni, zda je list prazdny
   public boolean isEmpty()  {
      return prvniUzel == null;
      }

   // tisk seznamu (list)
   public void print() {
      if ( isEmpty() ) {
           System.out.printf( "Empty %s\n", jmeno );
           return;
        }

       System.out.printf( "The %s is: ", jmeno );
       Uzel current = prvniUzel;
       // rekurze
       while ( current != null ) {
            System.out.printf( "%s ", current.getData() );
            current = current.getDalsiUzel();
         }
       System.out.println( "\n" );
   }
}
```

Třída *SpojovySeznamTest* názorně ilustruje použití uvedeného spojového seznamu.

Třída SpojovySeznamTest

```
public class SpojovySeznamTest {
   public static void main(String[] args) {
       SpojovySeznam list = new SpojovySeznam();

   // vkladani celych cisel do seznamu
   list.insertAtFront( -1 );
   list.print();
   list.insertAtFront( 0 );
   list.print();
   list.insertAtBack( 1 );
   list.print();
   list.insertAtBack( 5 );
   list.print();

   // odstranovani prvku seznamu, tisk po kazde operaci
   try
   {
      Object removedObject = list.removeFromFront();
      System.out.printf( "%s odstranen\n", removedObject );
      list.print();

      removedObject = list.removeFromFront();
      System.out.printf( "%s odstranen\n", removedObject );
      list.print();
      removedObject = list.removeFromBack();
      System.out.printf( "%s odstranen\n", removedObject );
      list.print();

      removedObject = list.removeFromBack();
      System.out.printf( "%s odstranen\n", removedObject );
      list.print();
      } // end try

   catch ( EmptyListException emptyListException ) {
      emptyListException.printStackTrace();
      } // end catch
   }
}
```

Při odstraňování uzlů seznamu může nastat situace, že spojový seznam je prázdný, což vyhodí výjimku. Proto jsou v hlavním programu bloky try a catch.

12.3 Vytváření vlastních seznamů

Knihovny kolekcí nám poskytují bohatý výběr různých seznamů. Často ale potřebujeme z konkrétního knihovního seznamu využívat pouze některé metody a ostatní metody nechceme uživateli zpřístupnit. K vytváření specializovaných seznamů existují v podstatě dvě cesty. První využívá dědičnosti a druhý skládání resp. delegování. Ukážeme si obě možnosti a shrneme výhody a nevýhody jednotlivých postupů.

12.3.1 Využití dědičnosti při vytváření vlastních seznamů

Našim cílem je vytvořit např. zásobník při využití třídy *SpojovySeznam*, kterou jsme představili v úvodu této kapitoly. Nejjednodušší způsob jak je vidět je využití dědičnosti, což znamená, že třída zásobník (v tomto případě nazvaný *ZasobnikDedicnost*), bude podtřídou třídy *SpojovySeznam*. To, že je třída podtřídou jiné třídy znamená, že dědí všechny datové atributy a všechny metody nadtřídy, tedy třídy *SpojovySeznam*. U třídy *ZasobnikDedicnost* deklarujeme konstruktor a dvě typické metody pro zásobník, a to *push()* – pro vkládání prvku do zásobníku a metodu *pop()* pro odebrání prvku ze zásobníku.

Třída ZasobnikDedicnost

```
public class ZasobnikDedicnost extends SpojovySeznam {
   // bez parametricky konstruktor
   public ZasobnikDedicnost()  {
      super( "Zasobnik dedicnost" );
   }

   // pridani objectu do zasobniku
   public void push( Object object ) {
      insertAtFront( object );
   }

   // odebrani objektu ze zasobniku
   public Object pop() throws EmptyListException  {
      return removeFromFront();
   }
}
```

Třída *ZasobnikTest* ilustruje použití zásobníku. Při odebírání prvků ze zásobníku je využita výjimka *EmptyListException*, která hlídá okamžik, kdy je zásobník prázdný a přijde požadavek na odebrání dalšího prvku ze zásobníku.

Třída ZasobnikDedicnostTest

```
public class ZasobnikDedicnostTest {
   public static void main(String[] args) {
       ZasobnikDedicnost stack = new ZasobnikDedicnost();

      // use push method
      stack.push( -1 );
      stack.print();
      stack.push( 0 );
      stack.print();
      stack.push( 1 );
      stack.print();
      stack.push( 5 );
      stack.print();


      // odebrani prvku ze zasobniku
      try {
         Object odstranenyObjekt = null;
         while ( true ) {
           // pouziti metody pop()
            odstranenyObjekt = stack.pop();
            System.out.printf( "%s popped\n",
                               odstranenyObjekt );
            stack.print();
         }
      }
      catch ( EmptyListException emptyListException ) {
         emptyListException.printStackTrace();
      }
   } // end main
}
```

Vyhodnocení využití dědičnosti při deklarování specializovaných seznamů.

Výhody:

Zdánlivě rychlá konstrukce. Deklarujeme metody, které budeme využívat.

Nevýhody:

Jedná se o dědičnost, což znamená, že zdědíme všechny metody z nadtřídy. Z principů objektově orientovaného přístupu plyne, že můžeme používat všechny metody, tedy nově deklarované a všechny zděděné metody. Zděděné metody v našem případě znamenají odebírání jak z konce, tak ze začátku. Abychom zamezili používání „nepožadovaných“ metod musíme všechny tyto metody z nadtřídy zastínit v podtřídě.

Tedy deklarovat metodu se stejným jménem a počtem a typem argumentů metody a s prázdným tělem metody. To ale není úplně nejlepší řešení, uživatel nespustí nedovolené metody, ale zároveň se

nedoví nic o tom, zda proběhly metody správně nebo ne. Jiné možná lepší řešení je využití výjimek, které by uživateli vypsalo, že vyvolal nevhodnou metodu, ale že se nic nestalo, protože výjimka odchytila nevhodné řešení.

12.3.2 Využití skládání (delegování) při vytváření vlastních seznamů

Druhé řešení, které je více používané je využití skládání. V některých literaturách je toto řešení označováno jako využití delegování, tedy přesun pravomocí na někoho jiného. Delegujeme své pravomoci na svého kolegu v případě naší nepřítomnosti.

Při tomto druhém způsobu deklarujeme novou samostatnou třídu, v našem případě třídu *ZasobnikSkladani*. Datový atribut této třídy *ZasobnikSeznam* je typu *SpojovySeznam* a tento datový atribut bude fungovat jako námi požadovaný zásobník. Třída *ZasobnikSkladani* je tvořena (skládá se) z tohoto zásobníku. Kromě deklarace datového atributu musíme ještě doplnit konstruktor, ve kterém zabezpečíme, že proměnná *zasobnikSeznam* bude ukazovat na novou instanci třídy *SpojovyZasobnik* (tu jsme si vytvořili dříve) a její jméno bude *zasobnik*. Doplníme ještě metody pro vkládání do zásobníku, výběr ze zásobníku, metodu ověřující prázdnost zásobníku a metodu tisk. Když se podíváme dovnitř všech uvedených metod tak vidíme, že na proměnnou *zasobnikSeznam* aplikujeme některou z metod uvedených ve třídě *SpojovySeznam*.

Třída ZasobnikSkladani

```
public class ZasobnikSkladani {
   // pouziva se také termin delegovani
   private SpojovySeznam zasobnikSeznam; //datovy atribut
seznam

   // konstruktor bez parametru
   public ZasobnikSkladani() {
      zasobnikSeznam = new SpojovySeznam( "zasobnik" );
   }

   // pridani objectu do zasobniku
   public void push( Object object ) {
      zasobnikSeznam.insertAtFront( object );
   }

   // odebrani objectu ze zasobniku
   public Object pop() throws EmptyListException {
      return zasobnikSeznam.removeFromFront();
   }

   // zjisteni, zda je zasobnik prazdny
   public boolean isEmpty() {
      return zasobnikSeznam.isEmpty();
   }
```

```
// tisk obsahu zasobniku
   public void print()  {
      zasobnikSeznam.print();
   }
}
```

Nevýhody řešení:

Uvedený postup je trochu pracnější. Musíme deklarovat datový atribut a explicitně deklarovat všechny metody, které chceme používat.

Výhody řešení:

Řešení je bezpečnější a praktičtější v tom smyslu, že uživatel může např. u deklarovaného zásobníku využít pouze metody, které jsme deklarovali ve třídě *ZasobnikSkladani*. K jiným metodám, které jsme v této třídě nedeklarovali, nemá uživatel přístup. Nemusíme mít obavy, zda jsme předeklarovali všechny metody, či vše zabezpečili výjimkami.

Spojové seznamy tvoří důležitou součást datových struktur. Je dobré si uvědomit, které třídy je třeba navrhnout, aby vznikl spojový seznam. Jedná se o jednoduchou aplikaci s omezeným počtem metod. I přes to je aplikace plně funkční.

Jak by se změnila třída *Uzel*, pokud bychom pracovali s obousměrným spojovým seznamem?

Jaké další operace by obousměrný seznam umožňoval.

Jak bychom do spojového seznamu ukládali objektové typy?

13 Stromové struktury

V této kapitole se dozvíte:

- jak vytváříme stromové struktury v OOP,
- jaké třídy musíme navrhnout,
- jakou musí mít třídy funkčnost,
- jak se prakticky využívá rekurze.

Po jejím prostudování budete schopni:

- navrhnout a implementovat stromovou strukturu,
- prakticky využívat algoritmy rekurze.

Klíčová slova této kapitoly:

objekt, instance, datový atribut, třída

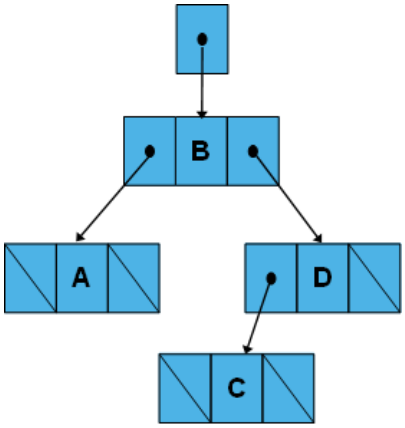
Strom je nelineární, dvoudimenzionální datová struktura se speciálními vlastnostmi. Spojové seznamy jsou lineární datové struktury. Uzly stromu obsahují dva (binární strom) nebo více odkazů na další uzly stromu. *Kořenový uzel (Root* node) je první uzel, počátek každého stromu. My se budeme zabývat pouze jednodušší variantou stromu, a sice binárním stromem.

Každá reference odkazuje na dítě (*child). Levé dítě* je prvním uzlem v levém podstromu (subtree). *Pravé dítě je prvním uzlem v pravém podstromu.* Děti v uzlech se nazývají sourozenci (*siblings).* Uzly, které nemají pokračování (děti) jsou uzly listů (*leaf nodes).*

Binární prohledávací strom (binary search tree) je speciální řazení uzlů, ve kterých se nevyskytují duplikáty hodnot uzlů (tím se myslí datový atribut uzlu, který buď přímo obsahuje hodnotu primitivního typu, nebo odkazuje na objektový typ). Při ukládání číselných hodnot platí, že hodnoty v levých podstromech jsou menší než hodnoty v pravých podstromech. Rozlišujeme celkem tří průchody (někdy se také používá pojem traverzování) takovým stromem.

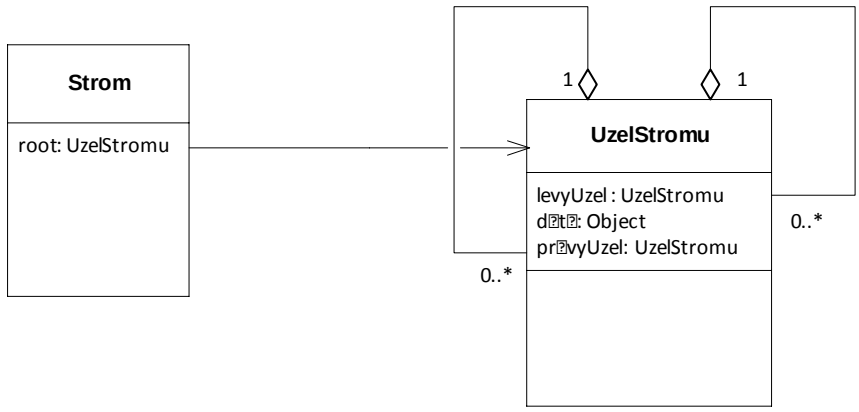
1. **Průchod inorder**
   Prochází levý podstrom, získá hodnotu uzlu, prochází pravý podstrom
2. **Průchod preorder**
   Získá datový atribut uzlu, prochází levý podstrom, prochází pravý podstrom

3. **Průchod postorder**
   Prochází levý podstrom, prochází pravý podstrom, získá datový atribut (hodnotu) uzlu



Obr. 13.1 Grafická struktura binárního stromu

Pokud chceme vytvořit aplikaci pro strukturu binárního stromu, musíme nejdříve deklarovat třídu *UzelStromu*, která deklaruje odpovídající datové atributy a metody a následně pak třídu *Strom*, která deklaruje potřebné datové atributy a metody.



Obr. 13.2 Diagram tříd binární stromové struktury.

13.1 Třída UzelStrom

Tato třída deklaruje tři datové atributy a to *levyUzel*, hodnotu (objekt) uložený v uzlu a *pravyUzel*. Jedná se o rekurzivní datovou strukturu, protože *levyUzel* a *pravýUzel* jsou typu *UzelStromu*. Ve třídě je dále deklarován konstruktor a přístupové metody k datovým atributům. Metoda *insert()* s argumentem hodnoty zabezpečí vytvoření nové instance třídy *UzelStrom* a její umístění do struktury stromu tak, aby vyhovovala požadavkům pro binární prohledávací strom. Tento požadavek zabezpečí, že do stromu nejsou ukládány stejné hodnoty

(nejsou povoleny duplikáty) a požadavek, že hodnoty v levých podstromech jsou menší než hodnoty v pravých podstromech.

Třída UzelStrom

```
public class UzelStromu {
   private UzelStromu levyUzel;
   private int data; // hodnota, objekt uzlu
   private UzelStromu pravyUzel;

   // konstruktor, inicializace datoveho atributu
   public UzelStromu( int dataUzlu ) {
      data = dataUzlu;
      levyUzel = pravyUzel = null; // uzly nemaji deti
   }

   public UzelStromu getLevyUzel() {
      return levyUzel;
   }
   public int getData() {
      return data;
   }

   public UzelStromu getPravyUzel() {
      return pravyUzel;
   }

   // nalezne bod pro vlozani a vlozi novy uzel;
   // ignoruje duplikovane hodnoty
   public void insert( int vkladanaHodnota ) {

      // vlozi do leveho podstromu
      if ( vkladanaHodnota < data ) {

         // vlozi novy UzelStromu
         if ( levyUzel == null )
            levyUzel = new UzelStromu( vkladanaHodnota );
         else
            // pokracuje v traverzovani leveho podstromu
            levyUzel.insert( vkladanaHodnota );
      } // end if

      //vkladani do praveho podstromu
      else if ( vkladanaHodnota > data ) {

         // vlozi new UzelStromu
         if ( pravyUzel == null )
            pravyUzel = new UzelStromu( vkladanaHodnota );
         else

            // pokracuje v traverzovani praveho podstromu
            pravyUzel.insert( vkladanaHodnota );
      } // end else if
   } // end metody insert
}
```

13.2 Třída Strom

Tato třída deklaruje jen jeden datový atribut a tím je odkaz na kořen stromu (root). Z tohoto bodu se odvíjí všechny požadované operace, tedy operace pro vložení nového uzlu stromu a operace průchodu stromem. Třída dále deklaruje konstruktor a metodu *vlozitUzel()*.

Průchod uzlem je možný třemi různými způsoby, které jsou deklarované v úvodu této kapitoly. Těmto průchodům odpovídají tři různé metody, a to *preorderTraversal()*, *inorderTraversal()* a *postordeerTraversal()*. Každá z těchto metod využívá ještě pomocnou metodu označenou jako *helper*. Tyto pomocné metody mají modifikátor private což znamená, že jsou volatelné z vnitřku třídy, ale nejsou dostupné z vnějšku třídy.

Třída Strom

```
public class Strom {
   private UzelStromu root;
   // konstruktor inicializuje prazdny strom celych cisel
   public Strom() {
      root = null;
   }

   // vlozi novy uzel do binarne prohledavaneho stromu
   public void vlozitUzel( int vkladanaHodnota ) {
      if ( root == null )
         // zde vytvori uzel root
         root = new UzelStromu( vkladanaHodnota );
      else
         root.insert( vkladanaHodnota );
   } // end metody vlozitUzel

   // zacatek pruchodu preorder
   public void preorderTraversal() {
      preorderHelper( root );
   }

   // recursivni metoda ktera provadi pruchod preorder
   private void preorderHelper( UzelStromu node ) {
      if ( node == null )
         return;

      // vystup dat.atributu uzlu
      System.out.printf( "%d ", node.getData());

      // pruchod levym podstromem
      preorderHelper( node.getLevyUzel() );

      // pruchod pravym podstromem
      preorderHelper( node.getPravyUzel() );
   } // konec metody preorderHelper()
```

```
// zacatek pruchodu inorder
   public void inorderTraversal() {
      inorderHelper( root );
   } // end metody inorderTraversal


   // recursivni metoda ktera provadi pruchod inorder
   private void inorderHelper( UzelStromu node ) {
      if ( node == null )
         return;

      // pruchod levym podstromem
      inorderHelper( node.getLevyUzel() );

      // vystup dat.atributu uzlu
      System.out.printf( "%d ", node.getData() );

      // pruchod pravym poduzlem
      inorderHelper( node.getPravyUzel() );
   }

   // zacatek pruchodu postorder
   public void postorderTraversal() {
      postorderHelper( root );
   }

   // rekursivni metoda pro pruchod postorder (traversal)
   private void postorderHelper( UzelStromu node ) {
      if ( node == null )
         return;

      // pruchod levym podstromem
      postorderHelper( node.getLevyUzel() );

      // pruchod pravym podstromem
      postorderHelper( node.getPravyUzel() );

      // vystup dat.atributu uzlu
      System.out.printf( "%d ", node.getData() );
   }
}
```

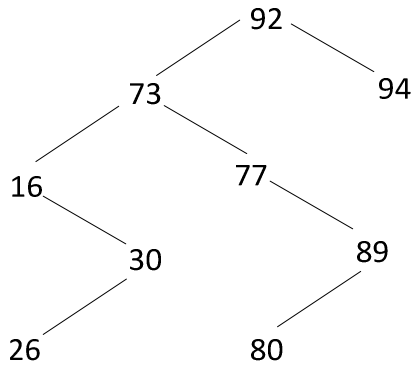
K ověření funkčnost je deklarovaná třída *StromTest*, která demonstruje využití struktury binárního stromu. Je možné vkládat náhodné hodnoty (tím myslíme datové hodnoty uzlu stromu), nebo doplnit program na vstup hodnot, které zadává uživatel.

Třída StromTest

```
import java.util.Random;
public class StromTest {
   public static void main(String[] args) {
       Strom tree = new Strom();
      int value;
      Random randomNumber = new Random();

      System.out.println("Inserting the following values: " );
```

```
// vloz 10 nahodnych celych cisel od 0-99 do stromu
      for ( int i = 1; i <= 10; i++ ) {
         value = randomNumber.nextInt( 100 );
         System.out.print( value + " " );
         tree.vlozitUzel( value );
      }

      System.out.println ( "\n\nPruchod preorder" );
      // vykona pruchod preorder stromem
      tree.preorderTraversal();

      System.out.println ( "\n\nPruchod inorder" );
      // vykona pruchod inorder stromem
      tree.inorderTraversal();

      System.out.println ( "\n\nPruchod postorder" );
      // vykona pruchod postorder stromem
      tree.postorderTraversal();
      System.out.println();
       }
}
```

Nechť máme zadanou řadu číselných hodnot, které vložíme do binárního stromu.
Řada hodnot: 92, 73, 77, 16, 30, 30, 94, 89, 26, 80. Pak bude mít vytvořený binární strom podle uvedených programů následující strukturu:



Obr. 13.3 Binární strom s celočíselnými hodnotami

Po spuštění hlavního programu dostaneme následující výsledky:

```
Pruchod preorder
92 73 16 30 26 77 89 80 94

Pruchod inorder
16 26 30 73 77 80 89 92 94
```

```
Pruchod postorder
26 30 16 80 89 77 73 94 92
```

Stromové struktury mají stále velké praktické uplatnění. Struktura binárních stromů je jednodušší než struktura stromů, ale i přesto jsou výhodné jak pro praktické využití, tak pro výukové účely. Podobně jako u spojového seznamu musíme nejdříve deklarovat třídy *UzelStromu* a pak třídu *Strom*. Musíme také stanovit způsob průchodu stromem.
Tato aplikace je také důležitá z pohledu praktického uplatnění rekurze. Práce se stromovou strukturou by bez rekurze byla značně komplikovaná a mnohdy nemožná.

Projděte kód metod a uvědomte si využívání rekurze.

Literatura

Pecinovský R.: Myslíme objektově v jazyku Java 5.0, Grada 2004

Zakhour S., a kol. Java 6 Výukový kurz. CPress 2007

Barnes, D., Kolling, M. Objects First with Java A Practical Introduction Using BlueJ. Pearson 4th edt. 2009

Arlow J., Neustadt I.: UML a unifikovaný proces vývoje aplikací. ComputerPress 2003

Polák, Merunka: Objektově orientované programování, Objektově orientované pojmy. Softwarové noviny 1993 série článků.

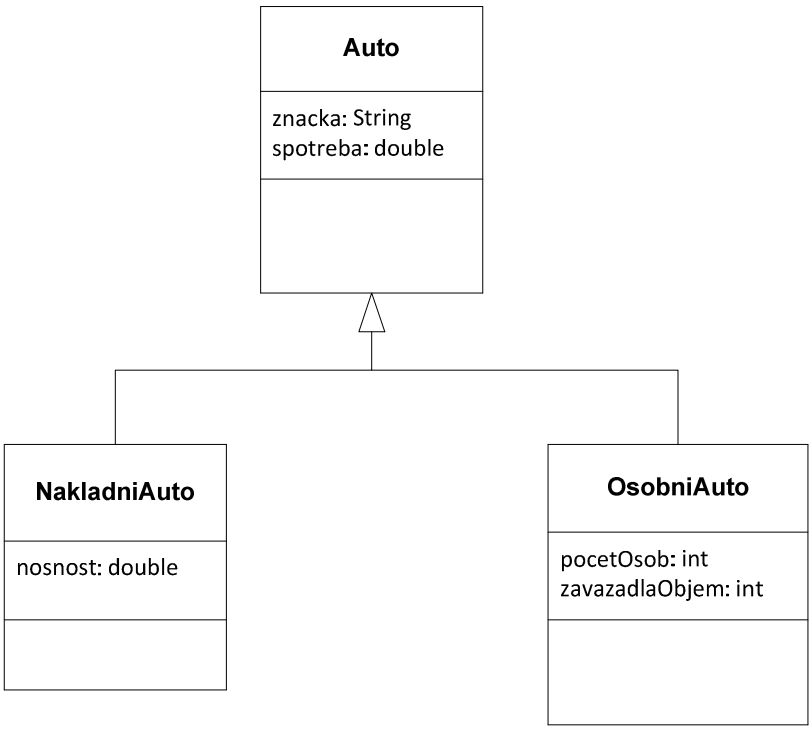
htpp://java. sun.com

http://www.osu.cz/hunka - zdrojové příklady a učební text

Příklady k samostatnému zpracování a odeslání emailem ke kontrole.

1. Vytvořte třídu *GrafickyObjekt*, která je složena (buď jako kompozice nebo agregace) z jednoho objektu třídy *Krizek* a jednoho objektu třídy *Obdelnik* viz čtvrtá kapitola. V zaslaném řešení nakreslete jednoduchý diagram tříd UML. Třída *GrafickyObjekt* má konstruktor s oběma objekty (krizek, obdelnik) jako parametry, přístupové a modifikační metody, metodu vykresli(), smaz(), posun(dx, dy), toString() a tisk(). Metoda vykresli() zobrazí instanci třídy *GrafickýObjekt* v kreslícím okně. Dále vytvořte třídu *GrafickyObjektTest*, která má hlavní metodu a provádí testování všech navržených metod.

2. Vytvořte třídu Auto s datovými atributy *značka* a *spotřeba* (počet litrů na 100 km). Doplňte konstruktor se všemi parametry, přístupové a modifikační metody, metodu toString() a tisk(). Dále vytvořte podtřídy *NákladniAuto* s datovým atributem nosnost a třídu OsobníAuto s datovými atributy počet osob, zavazadlaObjem. U obou podtříd doplňte konstruktory se všemi parametry, přístupové a modifikační metody a metodu toString(). Vytvořte třídy AutoTest, kde vytvoříte opd všech tříd instanci a ověříte všechny deklarované metody.

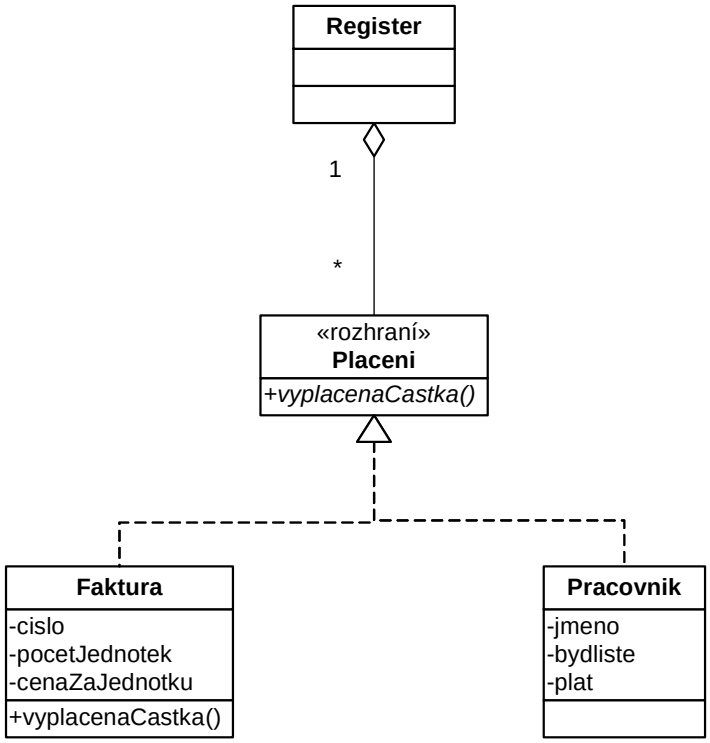


Obr. 14.1 Diagram tříd pro 2 korespondenční úkol

3. Vytvořte jednoduchou aplikaci pro zaměstnavatele, který platí jednak faktury a dále platí pracovníky měsíčním platem. Vytvořte proto rozhraní *Placeni*, které bude obsahovat metodu *vyplacenaCastka*. Toto rozhraní implementuje třída *Pracovník*, která má datové atributy *jmeno*, *bydliště* a *plat* a přístupové a modifikační metody k datovým atributům a metodu *vyplacenaCastka*, která pouze vrací datový atribut *plat*.

   Rozhraní *Placeni* implementuje také třída *Faktura*, která obsahuje datové atributy *cislo*, *pocetJednotek* a *cenaZaJednotku*. Dále třída obsahuje přístupové a modifikační metody a metodu *vyplacenaCastka*, která vynásobí *pocetJednotek* s *cenouZaJednotku* a tento výsledek vrátí v uvedené metodě.
   Objekty tříd *Faktura* a *Pracovník* budete ukládat do třídy *Register*, kterou si upravíte tak, aby byla schopna vkládat objekty typu *Placeni* (tedy rozhraní). Vytvořte třídu *PlaceníTest*, kde vytvoříte dva objekty třídy *Pracovník* a dva objekty třídy *Faktura* a uložte je

do objektu třídy *Register*. Všem objektům v registru zašlete zprávu placení, která vyvolá metodu placeni(), jejíž výsledek vytisknete. Struktura tříd je uvedena v diagramu tříd UML.



Obr. 14.2 Diagram tříd pro 3 korespondenční úkol