

5. Objektové modelování v Javě

- Kroky řešení problému na počítači - pár slov o SW inženýrství
- Organizace javových pg. - třídy, balíky
- „Objektovost“: zapouzdření, dědičnost
- Modifikátory přístupu k proměnným, metodám, třídám
- Deklarace import

Kroky řešení reálného problému na počítači

Generický (univerzální, obecný...) model postupu:

1. Zadání problému
2. Shromáždění informací o realitě a jejich analýza
3. Modelování reality na počítači a implementace požadovaných operací nad modelovanou realitou

Vývoj software je proces...

(podle JS, SW inženýrství):

1. při němž jsou **uživatelské potřeby**
2. transformovány na **požadavky na software**,
3. tyto jsou transformovány na **návrh**,
4. návrh je implementován pomocí **kódu**,
5. kód je **testován, dokumentován a certifikován** pro operační použití.

Celkový rámec vývoje SW

V tomto předmětu nás z toho bude zajímat jen něco a jen částečně:

1. **Specifikace** (tj. zadání a jeho formalizace)
2. **Vývoj** (tj. návrh a vlastní programování)
3. částečně **Validace** (z ní především testování)

•

1. **Specifikace SW**: Je třeba definovat funkcionalitu SW a operační omezení.
2. **Vývoj SW**: Je třeba vytvořit SW, který splňuje požadavky kladené ve specifikaci.
3. **Validace SW**: SW musí být validován („kolaudován“), aby bylo potvrzeno, že řeší právě to, co požaduje uživatel.
4. **Evoluce SW**: SW musí být dále rozvíjen, aby vyhověl měnícím se požadavkům zákazníka.

Metodiky vývoje SW

Tyto generické modely jsou dále rozpracovávány do podoby konkrétních **metodik**. Metodika (tvorby SW) je ucelený soubor inženýrských postupů, jak řízeným způsobem, s odhadnutelnou spotřebou zdrojů dospět k použitelnému SW produktu.

Některé skupiny metodik:

- strukturovaná
- objektová
- ...

Metodika typu "vodopád"

Nevracím se nikdy o více jak jednu úroveň zpět:

1. Analýza (Analysis)
2. Návrh (Design)
3. Implementace (Implementation)
4. Testování (Testing)
5. Nasazení (Deployment)

Nyní zpět k Javě a jednoduchým programům...

Srovnání Java - Pascal

Co bude odlišné oproti dosavadním programátorským zkušenostem?

Struktura a rozsah programu:

Pascal: program měl jeden nebo více zdrojových souborů (soubor = modul) tvořenými jednotlivými procedurami/fcemi, definicemi a deklaracemi (typů, proměnných...)

Java (a některé další OO jazyky): program je obvykle tvořen více soubory (soubor = popis jedné třídy) tvořenými deklaracemi metod a proměnných (případně dalších prvků) těchto tříd.

Organizace programových souborů

- **v Pascalu**: zdrojové (.pas) soubory, výsledný (jeden) spustitelný soubor (.exe), resp. přeložené kódy jednotek (.tpu)
- **v Javě**: zdrojové (.java) soubory, přeložené soubory v bajtkódu (.class) - jeden z nich spouštíme

Organizace zdrojových souborů

v Pascalu nebyla (nutná)

v Javě je nezbytná - zdrojové soubory organizujeme podle toho, ve kterých balících jsou třídy zařazeny

(přeložené soubory se **implicitně** ukládají vedle zdrojových)

Příklad - svět chovatelů a jejich psů

Zkusme naznačit, jak bychom realizovali jednoduchý systém, který bude

1. shromažďovat, ukládat a na požádání zpřístupňovat informace o psech (+ jejich výcviku, očkování...)
2. o jejich chovatelích
3. a dalších souvisejících entitách (např. chovatelských sdruženích, veterinářích,...)

Shromáždění informací o realitě

Zjistíme, jaké **typy objektů** se ve zkoumaném výseku reality vyskytují a které potřebujeme zachytit

člověk, pes, veterinář

Zjistíme a zachytíme vztahy mezi objekty našeho zájmu

člověk-chovatel vlastní psa

Zjistíme, které činnosti objekty (aktéři, aktoři) provádějí

veterinář psa očkuje, pes štěká, kousne člověka...

Jak zachytíme tyto informace

neformálními prostředky - tužkou na papíře vlastními slovy v přirozeném jazyce

formálně pomocí nějakého vyjadřovacího aparátu - např. grafického jazyka

pomocí CASE nástroje přímo na počítači

Zatím se přidržíme neformálního způsobu...

Modelování reality pomocí tříd

Určení základních **tříd**, tj.

skupin (kategorií) objektů, které mají podobné vlastnosti/schopnosti:

- Pes
- Clovek
- ...

Zápis třídy do zdrojového souboru

Soubor `Zivocich.java` bude obsahovat (pozor na velká/malá písmena - v obsahu i názvu souboru):

```
public class Zivocich { ... popis vlastností (proměnných, metod...)  
živočicha ...  
}
```

`public` značí, že třída je "veřejně" použitelná, tj. i mimo balík

Organizace tříd do balíků

Třídy zorganizujeme do balíků.

V balíku jsou vždy umístěny **související** třídy.

Co znamená související?

třídy, jejichž **objekty spolupracují** - **člověk, úřad**

třídy na podobné **úrovni abstrakce** - **chovatel, domácí zvíře**

třídy ze **stejné části reality** - **chovatel psů, pes**

Balíky

Balíky obvykle organizujeme do hierarchií, např.:

- svet
- svet.chovatelstvi
- svet.chovatelstvi.psi
- svet.chovatelstvi.morcata

Neplatí však, že by

- třídy "dceřinného" balíku (např. svet.chovatelstvi.psi)
- byly zároveň třídami balíku "rodičovského" (svet.chovatelstvi)!!!

Hierarchie balíků má tedy význam spíše pro srozumitelnost a logické členění.

Příslušnost třídy k balíku

Deklarujeme ji syntaxí: `package názevbalíku;`

- Uvádíme obvykle jako **první** deklaraci v zdrojovém souboru;
- Příslušnost k balíku musíme současně **potvrdit správným umístěním** zdrojového souboru do adresářové struktury;
- např. zdrojový soubor třídy `Pes` umístíme do podadresáře `svet\chovatelstvi\psi`
- Neuvedeme-li příslušnost k balíku, stane se třída součástí **implicitního balíku** - to však nelze pro jakékoli větší a/nebo znovupoužívané třídy či dokonce programy doporučit a zde nebude tolerováno!

Dědičnost

V realitě jsme často svědci toho, že třídy jsou **podtřídami** jiných:

- tj. všechny objekty podtřídy jsou zároveň objekty nadtřídy, např. každý objekt typu (třídy) `ChovatelPsi` je současně typu `Cllovek` nebo
- např. každý objekt typu (třídy) `Pes` je současně typu `DomaciZvire` (alespoň v našem výseku reality - existují i psi "nedomáci"...)

Podtřída je tedy "zjemněním" nadtřídy:

- přebírá její vlastnosti a zpravidla přidává další, **rozšiřuje** svou nadtřídu/předka

V Javě je **každá** uživatelem definovaná třída potomkem nějaké jiné - neuvedeme-li předka explicitně, je předkem vestavěná třída `Object`

Terminologie dědičnosti

Terminologie:

- Nadtřídě (superclass) se také říká "(bezprostřední) předek", "rodičovská třída"

- Podtřídě (subclass) se také říká "(bezprostřední) potomek", "dceřinná třída"

Dědění může mít i více "generací", např.

```
Zivocich <- Clovek <- Chovatel (živočich je rodičovskou třídou člověka, ten je rodičovskou třídou chovatele)
```

Přeneseně tedy předkem (nikoli bezprostředním) chovatele je živočich.

Jak zapisujeme dědění

Klíčovým slovem **extends**:

```
public class Clovek extends Zivocich {  
    ... popis vlastností (proměnných, metod...) člověka navíc oproti živočichovi ...  
}
```

Deklarace import NázevTřídy

Deklarace import nesouvisí s děděním, ale s organizací tříd programu do balíků:

- Umožní odkazovat se **v rámci kódu jedné třídy na ostatní třídy**
- Syntaxe: `import názevtřídy;`
- kde **názevtřídy** je uveden včetně názvu balíku
- Píšeme obvykle ihned po deklaraci příslušnosti k balíku (`package názevbalíku;`)

Import není nutné deklarovat mezi třídami téhož balíku!

Deklarace import *názevbalíku.**

Pak lze používat všechny třídy z uvedeného balíku

Doporučuje se "import s hvězdičkou" nepoužívat:

- jinak nevíme nikdy s jistotou, ze kterého balíku se daná třída použila;
- i profesionálové to však někdy používají :-)
- lze tolerovat tam, kde používáme z určitého balíku většinu tříd;
- v tomto úvodním kurzu většinou tolerovat nebudeme!

"Hvězdičkou" **nezpřístupíme třídy z podbalíků**, např.

- `import svet.* nepřístupní třídu svet.chovatelstvi.Chovatel`

Opakování - vlastnosti tříd

Jak víme, třídy popisují skupiny objektů podobných vlastností

Třídy mohou mít tyto skupiny **vlastností**:

- Metody - procedury/funkce, které pracují (především) s objekty této třídy
- Proměnné - pojmenované datové prvky (hodnoty) uchovávané v každém objektu této třídy

Vlastnosti jsou ve třídě "schované", tzv. **zapouzdřené** (encapsulated)

Třída připomíná pascalský záznam (record), ten však zapouzdřuje jen proměnné,

nikoli metody.

Dědičnost (alespoň v javovém smyslu) znamená, že dceřinná třída (podtřída, potomek)

- má **všechny** vlastnosti (metody, proměnné) nadtřídy
- + vlastnosti uvedené přímo v deklaraci podtřídy

Příklad

Cíl: vylepšit třídu `Ucet`

Postup:

1. Zdokonalíme náš příklad s účtem tak, aby si účet "hlídal", kolik se z něj převádí peněz
2. Zdokonalenou verzi třídy `Ucet` nazveme `KontokorentniUcet`

```
3. public class KontokorentniUcet extends Ucet {
4.     // double zustatek; znovu neuvádíme
5.     // ... zdědí se z nadtřídy/předka "Ucet"
6.
7.     // kolik mohu "jít do mínusu"
8.     double povolenyKontokorent;
9.
10.    public void pridej(double castka) {
11.        if (zustatek + povolenyKontokorent + castka >= 0) {
12.            // zavoláme původní "neopatrnou" metodu
13.            super.pridej(castka);
14.        } else {
15.            System.err.println("Nelze odebrat částku " +
16.                (-castka));
17.        }
18.    }
19.    // public void vypisZustatek() ... zdědí se
20.    // public void prevedNa(Ucet u, double castka) ... zdědí
21.    // ... předpokládejme, že v třídě "Ucet" používáme
22.    //     pridej(-castka);
23.    //     u.pridej(castka);
24.    // }
25.}
```

[Příklad kompletního zdrojového kódu třídy](#)

Vzorový zdroják sám o sobě nepůjde přeložit, protože nemáme třídu, na níž závisí. Celý kód vystavím až po kontrole příslušných úloh.

Příklad - co tam bylo nového

- Klíčové slovo `extends` - značí, že třída `KontokorentniUcet` je potomkem/podtřídou/rozšířením/dceřinnou třídou (**subclass**) třídy `Ucet`.
- Konstrukce `super.metoda(...)`; značí, že je volána metoda rodičovské třídy/předka/nadtřídy (**superclass**). **Kdyby se nevolala překrytá metoda, super by se neuvádělo.**
- Větvení `if() {...} else {...}` - složené závorky se používají k

uzavření příkazů do sekvence - ve smyslu pascalského begin/end.

Přístupová práva

Přístup ke třídám i jejím prvkům lze (podobně jako např. v C++) regulovat:

- Přístupem se rozumí jakékoli použití dané třídy, prvku...
- Omezení přístupu je kontrolováno hned při překladu -> není-li přístup povolen, nelze program ani přeložit.
- Tímto způsobem lze regulovat přístup staticky, mezi celými třídami, nikoli pro jednotlivé objekty
- Jiný způsob zabezpečení představuje tzv. **security manager**, který lze aktivovat při spuštění JVM.

Granularita omezení přístupu

Přístup je v Javě regulován **jednotlivě po prvcích**

ne jako v C++ po blocích

Omezení přístupu je určeno uvedením jednoho z tzv. **modifikátoru přístupu** (**access modifier**) nebo naopak **neuvedením žádného**.

Typy omezení přístupu

- Existují čtyři možnosti:
 - `public` = veřejný
 - `protected` = chráněný
 - **modifikátor neuveden** = říká se **lokální v balíku** nebo **chráněný v balíku** nebo "přátelský"
 - `private` = soukromý

Kde jsou která omezení aplikovatelná?

Pro **třídy**:

- veřejné - `public`
- neveřejné - lokální v balíku

Pro **vlastnosti tříd** = proměnné/metody:

- veřejné - `public`
- chráněné - `protected`
- neveřejné - lokální v balíku
- soukromé - `private`

Příklad - `public`

`public` => přístupné odevšad

```
public class Ucet { ... }
```

třída `Ucet` je veřejná = lze např.

- vytvořit objekt typu `Ucet` i v metodě jiné třídy

- deklarovat podtřidu třídy `Ucet` ve stejném i jiném balíku

Příklad - `protected`

`protected` => přístupné jen **z podtříd a ze tříd stejného balíku**

```
public class Ucet {
    // chráněná proměnná
    protected float povolenyKontokorent;
}
```

používá se jak pro metody (velmi často), tak pro proměnné (méně často)

Příklad - `přátelský`

lokální v balíku = přátelský => přístupné jen **ze tříd stejného balíku**, už ale ne z podtříd, jsou-li v jiném balíku

```
public class Ucet {
    Date created; // přátelská proměnná
}
```

- používá se spíše u proměnných než metod, ale dost často se vyskytuje z lenosti programátora, kterému se nechce psát `protected`
- osobně moc nedoporučuji, protože svazuje přístupová práva s organizací do balíků (-> a ta se může přece jen měnit častěji než např. vztah nadtřída-podtřída.)
- Mohlo by mít význam, je-li práce rozdělena na více lidí na jednom balíku pracuje jen jeden člověk - pak si může přátelským přístupem chránit své neveřejné prvky/třídy -> nesmí ovšem nikdo jiný chtít mé třídy rozšiřovat a používat přitom přátelské prvky.
- Používá se relativně často pro neveřejné třídy definované v jednom zdrojovém souboru se třídou veřejnou.

Příklad - `private`

`private` => přístupné jen v rámci třídy, ani v podtřídách - používá se častěji pro proměnné než metody

označením `private` prvek **zneviditelníme i případným podtřídám!**

```
public class Ucet {
    private String majitel;
    ...
}
```

- proměnná `majitel` je soukromá = nelze k ní přímo přistoupit ani v podtřídě - je tedy třeba zpřístupnit proměnnou pro "vnější" potřeby jinak, např.
- přístupovými metodami `setMajitel(String m)` a `String getMajitel()`

Když si nevíte rady

Nastavení přístupových práv k třídě pomocí modifikátorů se děje na úrovni tříd, tj. vztahuje se pak na **všechny objekty příslušné třídy** i na její **statické**

vlastnosti (proměnné, metody) atd.

Nastavení musí vycházet z povahy dotyčné proměnné či metody.

Nevíme-li si rady, jaká práva přidělit, řídíme se následujícím:

- metoda by měla být `public`, je-li užitečná i mimo třídu či balík - "navenek"
- jinak `protected`
- máme-li záruku, že metoda bude v případných podtřídách nepotřebná, může být `private` - **ale kdy tu záruku máme???**
- proměnná by měla být `private`, nebo `protected`, je-li potřeba přímý přístup v podtřídě
- téměř nikdy bychom neměli deklarovat proměnné jako `public`!

Přístupová práva a umístění deklarácí do souborů

- Třídy deklarované jako **veřejné** (`public`) musí být umístěné do souborů s názvem totožným s názvem třídy (+přípona `.java`) i na systémech Windows (vč. velikosti písmen)
 - kromě takové třídy však může být v tomtéž souboru i libovolný počet deklarácí neveřejných tříd
 - `private` nemají význam, ale **přátelské** ano
-