Implementace více rozhraní. Rozšiřování rozhraní. Statické a výchozí metody. Balíky. Viditelnost (přístupová práva).

October 14, 2014

1 Implementace více rozhraní současně

1.1 Implementace více rozhraní současně

Třída sice smí dědit maximálně z jedné nadtřídy, ale

- zato může současně implementovat libovolný počet rozhraní!
- Podmínkou ovšem je, aby se metody ze všech implementovaných rozhraní snesly v jedné třídě.
- Které že se nesnesou? Např. dvě metody se skoro stejnou hlavičkou, lišící se jen návratovým typem...

1.2 Implementace více rozhraní současně - příklad

Příklad - kromě výše uvedeného intf. *Informing* mějme ještě:

```
public interface Screaming {
    void scream();
}
```

Třída Person implementuje dvě rozhraní:

```
public class Person
    implements Informing, Screaming {
    ...
    public void writeInfo() {
        ...
    }
    public void scream() {
        ...
    }
}
```

2 Rozšiřování rozhraní

2.1 Rozšiřování rozhraní

Podobně jako u tříd, i rozhraní mohou být rozšiřována/specializována. Mohou dědit.Na rozdíl od třídy, která dědí maximálně z jedné nadtřídy –

- z rozhraní můžeme odvozovat potomky (podrozhraní - *subinterfaces*)
- dokonce i *vícenásobně* - z více rozhraní odvodíme společného potomka slučujícího a rozšiřujícího vlastnosti všech předků.

2.2 Rozšiřování rozhraní

Přesto to nepřináší problémy jako u klasické plné vícenásobné dědičnosti např. v C++, protože rozhraní samo:

- nemá proměnné
- metody neimplementuje
- nedochází tedy k nejednoznačnostem a konfliktům při podědění neprázdných, implementovaných metod a proměnných

2.3 Rozšiřování rozhraní – příklad

Příklad - *Informing* informuje jen trochu, *WellInforming* je schopen ke standardním informacím (*writeInfo*) přidat dodatečné informace (*writeAdditionalInfo*).

```
public interface Informing {
   void writeInfo();
}

public interface WellInforming extends Informing {
   void writeAdditionalInfo();
}
```

Třída, která chce implementovat intf. *WellInforming*, musí implementovat *obě* metody předepsané tímto rozhraním. Např.:

```
public class Informator implements WellInforming {
   public void writeInfo() {
      ... // kód metody
   }
   public void writeAdditionalInfo() {
      ... // kód metody
   }
}
```

2.4 Vícenásobné rozšiřování rozhraní – správně

```
interface A {
   void someMethod();
}
interface B {
   void someMethod();
}
interface AB extends A, B {
   // to je OK, someMethod má v obou případech
   // zcela shodnou signaturu (název, parametry, návratový typ)
}
```

2.5 Vícenásobné rozšiřování rozhraní – podruhé OK

```
interface A {
   void someMethod();
}
interface B {
   void someMethod(int param);
}
interface AB extends A, B {
   // to je OK, someMethod je pokaždé jiná, má
   // odlišné parametry, jde o běžné přetěžování metod
}
```

2.6 Vícenásobné rozšiřování rozhraní – špatně

```
interface A {
   void someMethod();
}
interface B {
   int someMethod();
}
interface AB extends A, B {
   // to není OK, someMethod má pokaždé
   // různý návratový typ při stejném názvu, parametrech
}
```

2.7 Rozhraní - poznámky

- Používají se i prázdná rozhraní - nepředepisující žádnou metodu
- deklarace, že třída implementuje také rozhraní, ji "k ničemu nezavazuje", ale poskytuje typovou informaci o dané třídě
- i v Java Core API jsou taková rozhraní - např. `java.lang.Cloneable`

3 Výchozí (implicitní) a statické metody

3.1 Výchozí (implicitní) metody – default methods

- Přidávají možnost implementovat přímo v rozhraní i určitou funkcionalitu.
- Tuto funkcionalitu je možné samozřejmě definovat jen na základě volání (dosud) abstraktních
- nebo statických (viz dále) metod tohoto rozhraní.
- V definici rozhraní jsou uvozeny klíčovým slovem default a obsahují rovněž tělo (tj. implementaci) metody.

3.2 Příklad

[zdroj: Java SE 8's New Language Features, Part 1: Interface Default/Static Methods and Lambda Expressions (`http://www.informit.com/articles/article.aspx?p=2191423`)]

```
public interface Addressable {
   String getStreet();
   String getCity();

   default String getFullAddress() {
      return getStreet() +", " + getCity();
   }
}
```

3.3 Významná použití výchozích metod

Výchozí metody se mohou zdát zbytečností... až na několik situací, kdy se velmi hodí:

- Vývoj existujících rozhraní: Dříve, bez výchozích metod, nebylo možné přidat v budoucí verzi rozhraní do něj metodu, aniž by se porušila binární kompatibilita se stávajícím kódem – všechny třídy implementující toto rozhraní by musely od nové verze implementovat i novou přidanou metodu, tudíž by stávající kód přestal fungovat.
- Zvýšení pružnosti návrhu: Výchozí metody nás zbavují časté nutnosti použít abstraktní třídu pro implementaci obecných metod, které by se jinak ve (často všech) implementujících neabstraktních třídách opakovaly. Abstraktní třída pak návrháře nutí de facto nikoli k pouhé implementaci rozhraní, ale k dědění z takové abstraktní třídy, což narušuje javový koncept preferující implementaci rozhraní před dědičností.

3.4 Rozšiřování rozhraní s výchozí metodou

Mějme rozhraní A obsahující nějakou výchozí metodu. Definujeme-li nyní rozhraní B jako rozšíření (extends) rozhraní A, mohou nastat tři různé situace:

- Jestliže výchozí metodu v rozhraní B nezmiňujeme, pak se podědí z A.

- V rozhraní B znovu uvedeme tuto metodu, ale jen její hlavičku (ne tělo). Pak ji nepodědíme, stane se abstraktní jako u každé obyčejné metody v rozhraní a každá třída implementující rozhraní B ji musí sama implementovat.
- V rozhraní B implementujeme metodu znovu, čímž se původní výchozí metoda překryje – jako při dědění mezi třídami.

3.5 Implementace více rozhraní s výchozími metodami – chybně

Následující kód Java 8 (a samozřejmě ani žádná starší) nezkompiluje:

```
interface A {
   default void someMethod() { /*bla bla*/ }
}
interface B {
   default void someMethod() { /*bla bla*/ }
}
class C implements A, B {
   // překladač by nevěděl, kterou someMethod() použít
}
```

3.6

javac InvalidDefaultMethods.java InvalidDefaultMethods.java:24: error: class C inherits unrelated defaults for someMethod() from types A and B

3.7 Implementace více rozhraní s výchozími metodami – správně

Následující kód Java 8 (ale samozřejmě žádná starší) bez potíží zkompiluje:

```
interface A {
   default void someMethod() { /* bla bla */ }
}
interface B {
   default void someMethod() { /* bla bla */ }
}
class D implements A, B {
   // překryjeme-li (dvojitě) poděděnou metodu, není problém
   // překladač nemusí přemýšlet, kterou default void someMethod() použít
   public void someMethod() {
      // the right stuff, this will be used
   }
}
```

3.8 Implementace více rozhraní s výchozími metodami – abstraktní a neabstraktní metoda

Následující kód Java 8 opět nezkompiluje: jedno rozhraní default metodu má a druhé ne.

```
interface A {
   void someMethod();
}
interface B {
   default void someMethod() {
      // whatever
   }
}
class E implements A, B {
   // nepřeloží, protože zůstává otázka:
   // má či nemá překladač použít výchozí metodu?
}
```

3.9

javac InvalidDefaultMethods.java

InvalidDefaultMethods.java:22: error: E is not abstract and does not override abstract method someMethod() in A

3.10 Rozšiřování více rozhraní s výchozími metodami – abstraktní a neabstraktní metoda

Následující kód Java 8 opět nezkompiluje: jedno rozhraní default metodu má a druhé ne.

```
interface A {
   void someMethod();
}
interface B {
   default void someMethod() {
      // whatever
   }
}
interface AB extends A, B {
   // ze stejného důvodu nepřeloží, protože zůstává otázka:
   // má či nemá překladač chápat toto rozhraní
   // tak, že obsahuje výchozí metodu?
}
```

3.11 Dokumentace

- Oracle The Java Tutorial: Default Methods (`http://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`)

4 Organizace tříd do balíků

4.1 Zápis třídy do zdrojového souboru

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

```
public class Person {
   ... popis vlastností (proměnných, metod...) osoby ...
}
```

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

4.2 Organizace tříd do balíků

Třídy zorganizujeme do balíků.V balíku jsou vždy umístěny *související* třídy.Co znamená související?

- třídy, jejichž **objekty spolupracují**
- třídy na podobné **úrovni abstrakce**
- třídy ze **stejné části reality**

4.3 Balíky

Balíky obvykle organizujeme do hierarchií, např.:

- `cz.muni.fi.pb162`
- `cz.muni.fi.pb162.banking`
- `cz.muni.fi.pb162.banking.credit`

Neplatí však, že by

- třídy "dceřinného" balíku (např. `cz.muni.fi.pb162.banking.credit`)
- byly zároveň třídami balíku "rodičovského" (`cz.muni.fi.pb162.banking`)!!!

Hierarchie balíků má tedy význam spíše pro srozumitelnost a logické členění. Jinak rozhoduje pouze relace "je ve stejném balíku" či "není ve stejném balíku".

4.4 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 `Person` umístíme do podadresáře `cz/muni/fi/pb162`

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!

4.5 Deklarace import názevbalíku.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 názevbalíku.NázevTřídy;`
- kde *název balíku* je následovaný *názvem třídy*
- Píšeme obvykle ihned po deklaraci příslušnosti k balíku (`package názevbalíku;`)

Import není nutné deklarovat mezi třídami téhož balíku!

4.6 Deklarace import názevbalíku.*

Pak lze používat všechny třídy z uvedeného balíkuDoporučuje se "import s hvězdičkou" nepoužívat vůbec:

- 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" *stejně nezpřístupníme třídy z podbalíků*, např.

- `import cz.*` nezpřístupní třídu `cz.muni.fi.pb162.Person`

5 Viditelnost (práva přístupu)

5.1 Viditelnost (práva přístupu)

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

- Přístupem se rozumí jakékoli použití dané třídy, prvku... prostě už vůbec výskyt daného identifikátorů tím může být omezen.
- Omezení přístupu/viditelnosti je kontrolováno hned při překladu → není-li přístup povolen, nelze program ani přeložit.
- Tímto způsobem lze regulovat přístup/viditelnosti staticky, mezi celými třídami, nikoli pro jednotlivé objekty!
- Jiný způsob zabezpečení představuje tzv. *security manager*, který lze aktivovat při spuštění JVM.

5.2 Granularita viditelnosti/omezení přístupu

- Přístup/viditelnost je v Javě regulován/a *jednotlivě po prvcích* (metoda, atribut),
- nikoli jako v C++ po blocích.
- Omezení přístupu/viditelnosti je určeno uvedením jednoho z tzv. *modifikátoru přístupu* (*access modifier*) nebo naopak *neuvedením žádného.*

5.3 Typy omezení viditelnosti/přístupu

Existují čtyři možnosti:

- `public` = veřejný
- `protected` = chráněný
- *modifikátor neuveden* = říká se *lokální v balíku* nebo *chráněný v balíku* evt. "přátelský"
- `private` = soukromý

5.4 Kde jsou která omezení aplikovatelná?

Třídy mohou být:

- veřejné – **public**
- neveřejné – lokální v balíku

Vlastnosti tříd = proměnné/metody mohou být:

- veřejné – **public**
- chráněné – **protected**
- neveřejné – lokální v balíku
- soukromé – **private**

5.5 Příklad – public

`public`, tj. viditelné/přístupné odevšad

```
public class Account {
   ...
}
```

třída `Account` je veřejná = lze např.

- vytvořit objekt typu `Account` i v metodě jiné třídy
- deklarovat podtřídu třídy `Account` ve stejném i jiném balíku

5.6 Příklad – protected

`protected`, tj. přístupné jen *z podtříd* a *ze tříd stejného balíku*

```
public class Account {
   // chráněná proměnná
   protected float creditLimit;
}
```

používá se pro metody (občas) a proměnné (málokdy)

5.7 Příklad – lokální v balíku

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

```
public class Account {
   Date created; // proměnná lokální v balíku
}
```

5.8 Kdy se lokální v balíku použije

- používá se málo, a to spíše u proměnných než metod, ale dost často se vyskytuje z lenosti programátora, kterému se nechce psát `protected` či `private`
- osobně moc nedoporučuji, protože svazuje viditelnost/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 jeho třídy rozšiřovat a používat přitom přátelské prvky.
- Používá se ale relativně často pro neveřejné třídy definované v jednom zdrojovém souboru se třídou veřejnou.

5.9 Příklad – private

`private`, tj. viditelné/přístupné jen v rámci třídy, ani v podtřídách - používá se častěji pro proměnné než metodyoznačením private prvek *zneviditelníme i případným podtřídám*!

```
public class Account {
   private String owner;
   ...
}
```

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

5.10 Když si nevíte rady

Nastavení viditelnosti 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. Obvykle se řídíme následujícím:

- metoda by měla být `public`, je-li užitečná i mimo třídu či balík – "navenek"
- Proměnná by měla být `private` a zcela výjimečně `protected` tehdy, je-li potřeba přímý přístup v podtřídě.

- jinak `private`
- Je metoda určená/vhodná k překrytí případných podtřídách? pak `protected`
- Téměř nikdy bychom neměli deklarovat proměnné jako `public` (vyjma případů, kdy jde o konstanty určené ke sdílení vně)!

5.11 Viditelnost a umístění deklarací 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 deklarací neveřejných (package-local, "přátelských") tříd