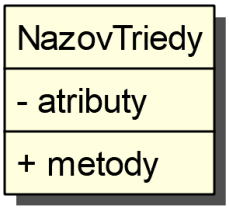
Informatika pre manažérov 2
Sprievodca cvičením 3 – Základy modelovania v UML
Autor: Marek Kvet

Z celého spektra UML diagramov sa na tomto cvičení obmedzíme len na *class diagram*, pomocou ktorého popisujeme *triedy* a *vzťahy* medzi nimi.
Všeobecný postup pri tvorbe a návrhu tried je nasledovný:

1. **abstrakcia** (zjednodušenie) reálneho objektu - slovná textová charakteristika inštancií triedy
2. **grafické znázornenie** pomocou diagramu tried v nejakom grafickom editore, napríklad v UML.FRI
3. **definícia triedy** v programovacom jazyku Java, napríklad pomocou nástroja BlueJ.

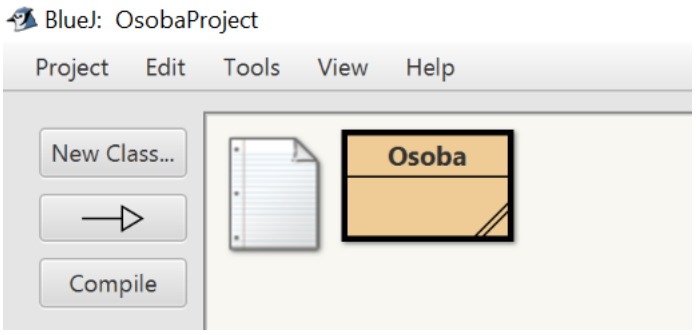
Začneme bodom 2 - grafickým znázornením existujúcich tried. K abstrakcii a návrhu vlastných tried sa dostaneme neskôr. Programovať na cvičení 3 nebudeme.

Class diagram zobrazuje triedu v obdĺžniku, ktorý pozostáva z troch častí, ako je znázornené na obrázku (UML diagram viete nakresliť v ľubovoľnom grafickom editore, napríklad aj vo Worde, ja používam UML.FRI). Vrchná časť obsahuje len *názov triedy*, v strednej sa udáva súkromná (private) časť triedy, teda *atribúty*, v spodnej časti sú verejné časti, teda *metódy*.



Ukážeme si to na konkrétnom príklade a podrobne rozoberieme.

Otvorte si adresár IpM_cv3_01. Nájdete v ňom podadresár OsobaProject, samostatný súbor OsobaDiagram.frip2 (výsledný diagram tried nakreslený v nástroji UML.FRI – k tomuto sa chceme prepracovať) a obrázok UML_Osoba.png (exportovaný obrázok z nástroja UML.FRI). Otvorte si projekt OsobaProject v BlueJ. Obsahuje jednu triedu, mali by ste vidieť toto:



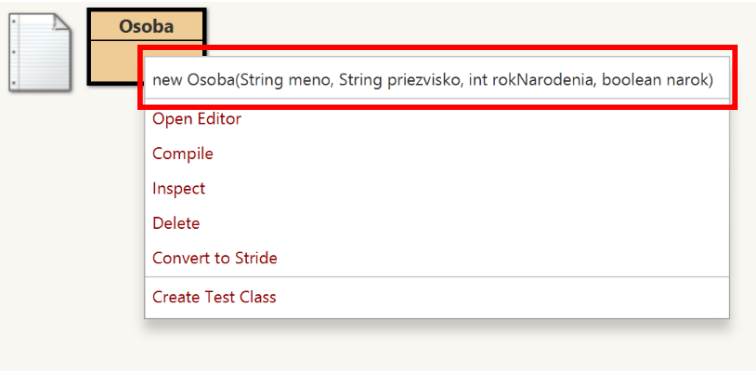
Ako sme spomínali na prednáške, existujú dva pohľady na triedu:

- **vonkajší** – slúži najmä pre používateľov, okrem názvu triedy zobrazuje *rozhranie*, teda zoznam správ, ktoré môžeme objektom posielať. V rozhraní sa nachádzajú len verejné

(public) správy. Správy, ktoré neposielame objektom, ale triede, sú v tomto diagrame podčiarknuté.

- **vnútorný** – kde vidíme všetko (atribúty, metódy, konštruktory). Slúži skôr pre programátorov, pre používateľov je niekedy príliš podrobný.

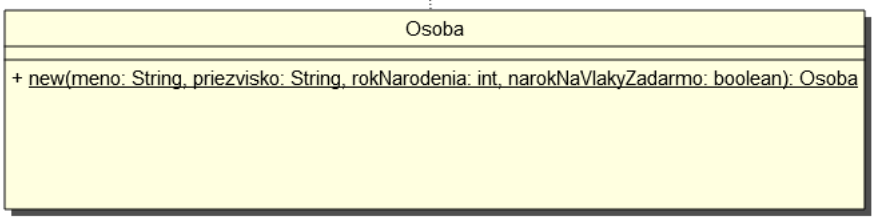
Začnime **vonkajším** pohľadom (je jednoduchší a kratší). Tu súkromné *atribúty* neriešime.
Aké správy môžeme poslať triede Osoba? Kliknite pravým tlačidlom myši na triedu a zobrazí sa Vám kontextové menu. V ňom uvidíte zoznam správ, ktoré posielame triede.



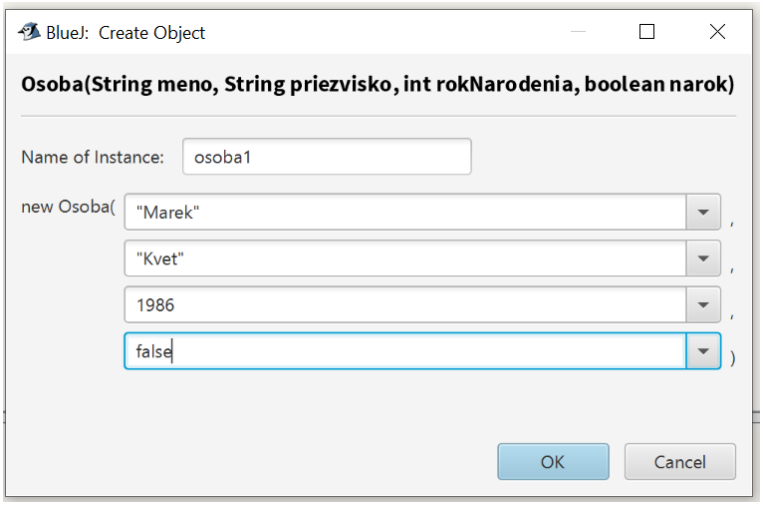
Triede *Osoba* môžeme poslať iba správu *new*, ktorá má 4 parametre (meno, priezvisko, rok narodenia a nárok na vlaky zadarmo). Návratovým typom takejto správy je referencia na objekt, teda môžeme povedať, že návratovým typom správy *new* je *Osoba*. Keďže túto správu posielame triede, bude príslušný riadok vo vonkajšom pohľade na triedu *Osoba* podčiarknutý. Správa je verejná (vo vonkajšom pohľade udávame len verejné informácie o objektoch), preto ju zapíšeme do spodnej časti UML diagramu a bude mať znamienko +. Zapíšeme to takto (syntakticky je to naopak ako v Jave – najprv sa udáva názov premennej, potom dvojbodka a potom typ premennej), konkrétne:

```
+ new(meno: String, priezvisko: String, rokNarodenia: int, narok: boolean): Osoba
```

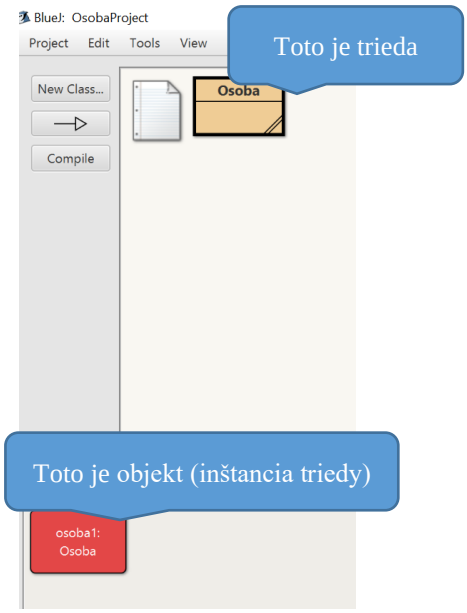
UML diagram po prvom kroku (stredná časť vyhradená pre atribúty zostáva prázdna):



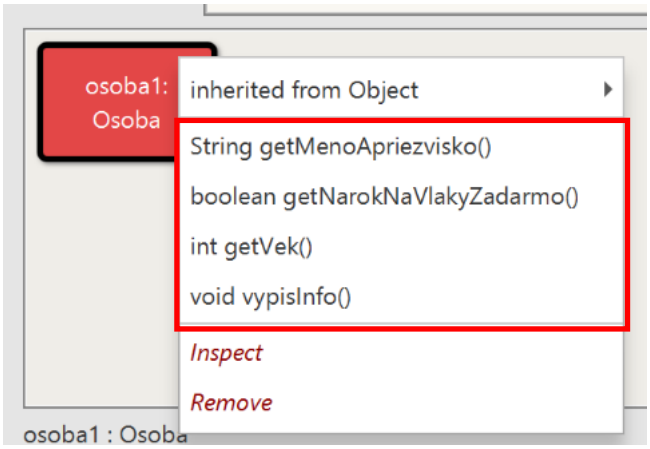
Aby sme videli, aké správy môžeme posielať objektom triedy *Osoba*, jeden objekt si vytvoríme. Pošleme triede správu *new*. Ak chceme zadávať parametre textového typu (String), musíme ich zapísať v úvodzovkách. Teda:
Marek – toto je nesprávne!
“Marek” – toto je OK
Parametre typu *boolean* môžu nadobúdať len hodnoty *true* (pravda) a *false* (nepravda). Správne vyplnená tabuľka parametrov môže vyzerať napríklad takto:



Po vytvorení inštancie triedy *Osoba* (objektu), dostaneme:



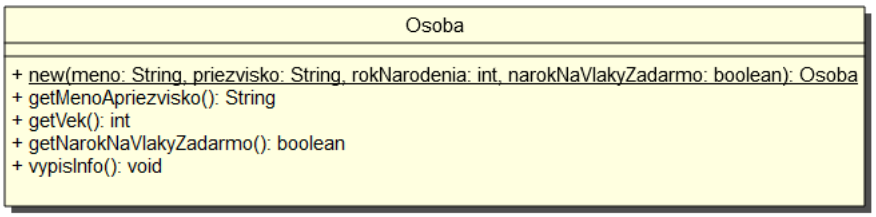
Ak chceme vidieť *rozhranie* objektu (zoznam správ, ktoré môžeme posielať objektu), klikneme pravým tlačidlom myši na objekt.



Objektom triedy *Osoba* môžeme poslať 4 rôzne správy. Preto do UML diagramu pridáme 4 riadky. Keďže tieto 4 správy posielame objektu a nie triede, riadky nepodčiarkneme. Správy sú verejné a preto budú mať na začiatku znak +. Syntax je rovnaká ako pri správe *new*. Do diagramu teda dopíšeme:

```
+ getMenoApriezvisko(): String
+ getVek(): int
+ getNarokNaVlakyZadarmo(): boolean
+ vypisInfo(): void
```

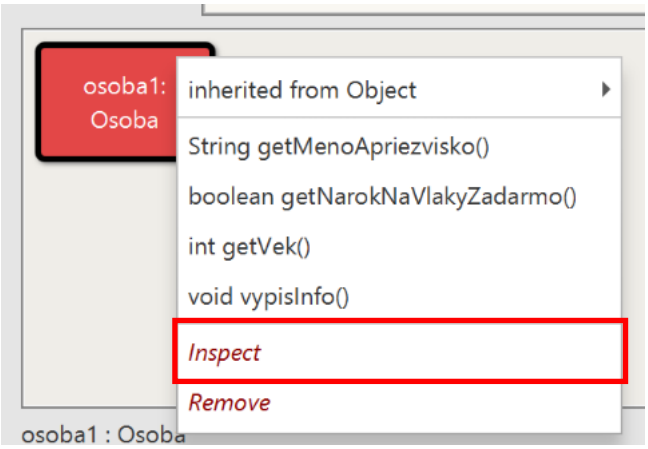
Výsledný UML diagram reprezentujúci **vonkajší** pohľad na triedu *Osoba* bude vyzerať takto:



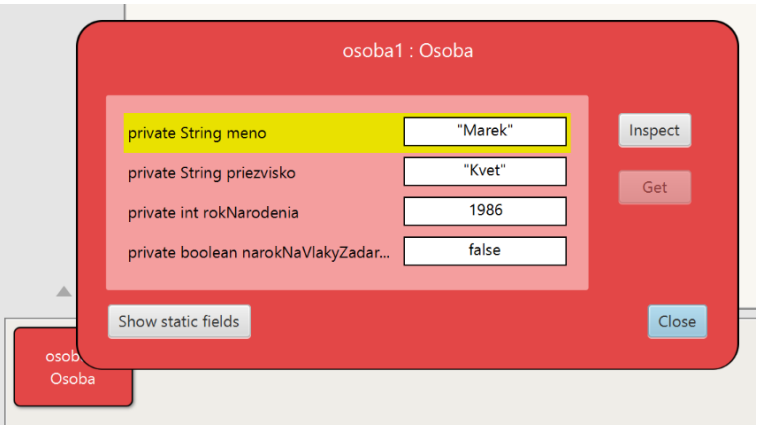
Na zopakovanie: Toto je **vonkajší** pohľad na triedu *Osoba*. Vo vonkajšom pohľade vidíme *rozhranie*, teda zoznam správ, ktoré môžeme objektom posielať. V rozhraní sa nachádzajú len **verejné** (public) správy. Správy, ktoré neposielame objektom, ale triede, sú v tomto diagrame podčiarknuté.

Pokračujeme **vnútorným** pohľadom. V ňom vidíme všetko tak, ako to vidí programátor. Začneme atribútmi. Aké vlastnosti (*atribúty*) má objekt triedy *Osoba*?

Ak nechceme študovať kód a hľadať v ňom atribúty, môžeme využiť možnosť *Inspect*, ktorú nám ponúka vývojové prostredie BlueJ. Stačí pravým tlačidlom myši kliknúť na objekt.



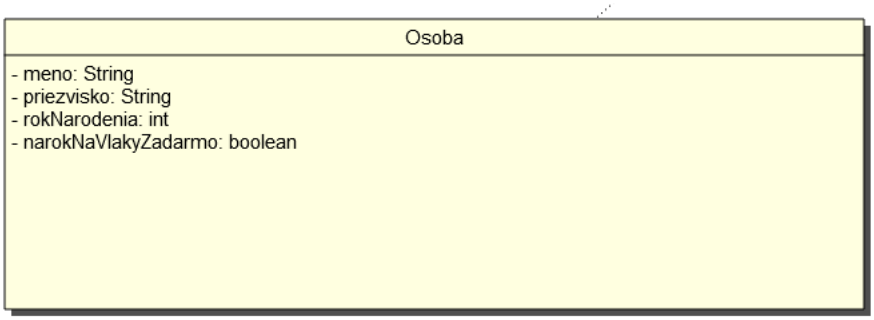
Keď zvolíme túto možnosť, uvidíme všetky *atribúty* objektu spolu s ich aktuálnymi hodnotami, teda *stav* objektu.



Vidíme, že objekt triedy *Osoba* má 4 atribúty. Všetky sú súkromné (private). V opačnom prípade by bol porušený jeden zo základných princípov objektového programovania, tzv. *zapuzdrenie*. Atribúty do strednej časti UML diagramu zapíšeme syntakticky takto:

```
-  atribút: typ
```

Atribúty sú súkromné, preto majú na začiatku znak -. UML diagram (vnútorný pohľad na triedu *Osoba*) po pridaní atribútov by mohol vyzerať takto:



Keď sa pozriete do kódu triedy *Osoba*, uvidíte, ako jednotlivé *atribúty* zapísané v kóde korešpondujú zápisu v diagrame. Znamienko – odpovedá kľúčovému slovu *private*.

```
public class Osoba {
    private String meno;
    private String priezvisko;
    private int rokNarodenia;
    private boolean narokNaVlakyZadarmo;
```

Teraz pridáme do UML diagramu ešte *metódy*. Vo **vnútornom** pohľade na triedu nebudeme hovoriť o *správach*, ktoré posielame (to robíme pri vonkajšom pohľade), ale o *metódach*. Preto budú v zobrazovaní oproti vonkajšiemu pohľadu rozdiely, najmä v zápise konštruktora.
Vo **vnútornom** pohľade nebudeme písať *new*, pretože nikde v kóde nič také nie je. Konštruktor v kóde vyzerá takto:

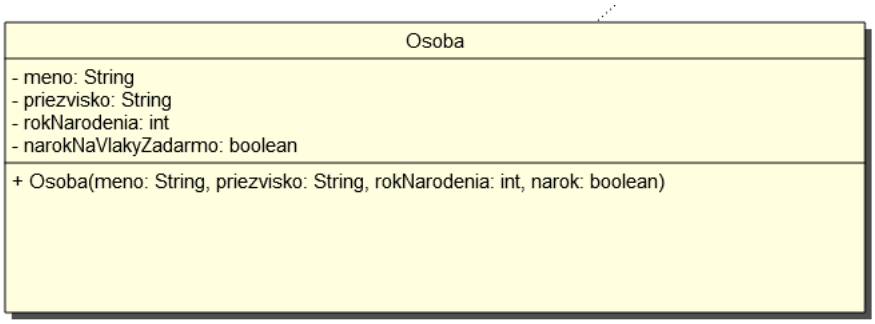
```java
public Osoba(String meno, String priezvisko, int rokNarodenia, boolean narok) {
    this.meno = meno;
    this.priezvisko = priezvisko;
    this.rokNarodenia = rokNarodenia;
    this.narokNaVlakyZadarmo = narok;
}
```

Na konštruktor sa pozeráme ako na špeciálnu verejnú (public) metódu, ktorá:

1. Nemá návratový typ (za slovíčkom *public* nie je *int, void, String*, ani nič podobné).
2. Má rovnaký názov ako trieda, teda *Osoba*.
3. Môže (nemusí) mať vstupné parametre, v našom prípade má 4.

V tomto duchu ho do diagramu zapíšeme rovnako ako metódu vo vonkajšom pohľade, teda nasledovne:

+ Osoba(String meno, String priezvisko, int rokNarodenia, boolean narok)

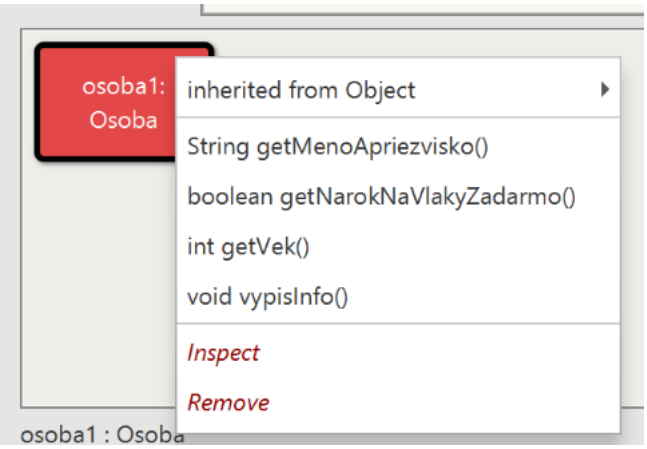


Nič nepodčiarkujeme, pretože z pohľadu *metód* nerozlišujeme, komu je určená, či triede alebo objektu. To robíme pri správach vo vonkajšom pohľade. Zobrazenie metódy v diagrame odpovedá jej hlavičke v kóde. Znak + odpovedá kľúčovému slovu *public*. Metódy zapisujeme rovnako ako vo vonkajšom pohľade. Správy (s výnimkou konštruktora) odpovedajú hlavičkám metód.

Treba si ale dať pozor na jednu vec, ktorá sa vyskytuje aj v triede *Osoba*. Kým vo **vonkajšom** pohľade na triedu vidíme *rozhranie*, teda zoznam **verejných** správ, ktoré môžeme poslať triede alebo objektu odkiaľkoľvek (zvonka), pri vnútornom pohľade musíme brať do úvahy aj také metódy, ktoré verejné nie sú. Všimnite si tento kúsok kódu:

```java
// Toto je súkromná správa, ktorú môže objekt poslať len sám sebe, zvonka ju poslať nemožno!
private int zistiAktualnyRok() {
    return Calendar.getInstance().get(Calendar.YEAR);
}
```

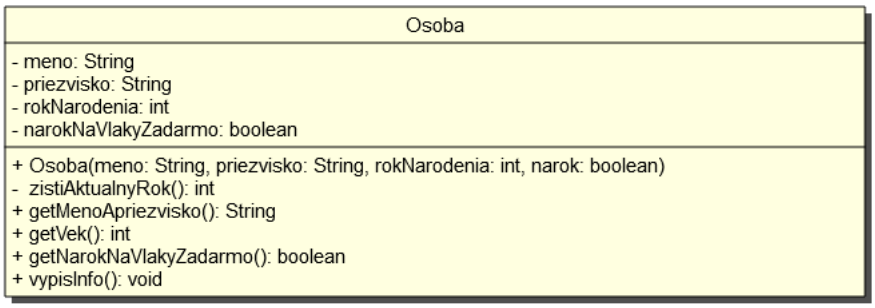
V triede *Osoba* sa vyskytuje metóda, ktorá je označená kľúčovým slovom *private*. Je teda súkromná. Správu *zistiAktualnyRok()* sme doteraz v rozhraní nevideli, pripomeňme si obrázok:



Takúto správu si môže objekt poslať iba sám sebe, zvonka je nedostupná. Vo **vnútornom** pohľade na triedu však aj táto metóda figurovať bude. Bude zaradená medzi metódami, teda v spodnej časti diagramu, ale bude mať na začiatku znamienko -, ktoré odpovedá súkromným častiam triedy. Niekedy sa stáva, že potrebujeme nejaký zložitejší proces (algoritmus) rozbiť na menšie časti (logické celky) a tak hlavnú metódu rozdelíme na viac menších. Tieto menšie časti bývajú súkromné, aby sa samostatne zvonku zavolať nedali. V našom prípade to využívame v metóde *getVek()*. Ako by ste určili vek osoby? Predpokladám, že takto:

```
2020 – rokNarodenia
```

Je to však správne len čiastočne. Mohol by som tento kód použiť aj o rok? O dva? Aktuálny rok sa zmení a vtedy by sme už nedostali správny vek osoby. Potrebujeme teda metódu, ktorá nám vráti aktuálny rok. Ale je to len vnútorná vec osoby, zvonka to praktický zmysel nemá. Záver: *V ojedinelých prípadoch môžu byť metódy aj súkromné, vo vnútornom pohľade na triedu treba zohľadniť aj tie, vo vonkajšom nie.*
Ostatné metódy do diagramu zapíšeme rovnako, ako sme si to ukázali vo vonkajšom pohľade. Výsledný UML diagram s **vnútorným** pohľadom na triedu *Osoba* by mohol vyzerať takto.



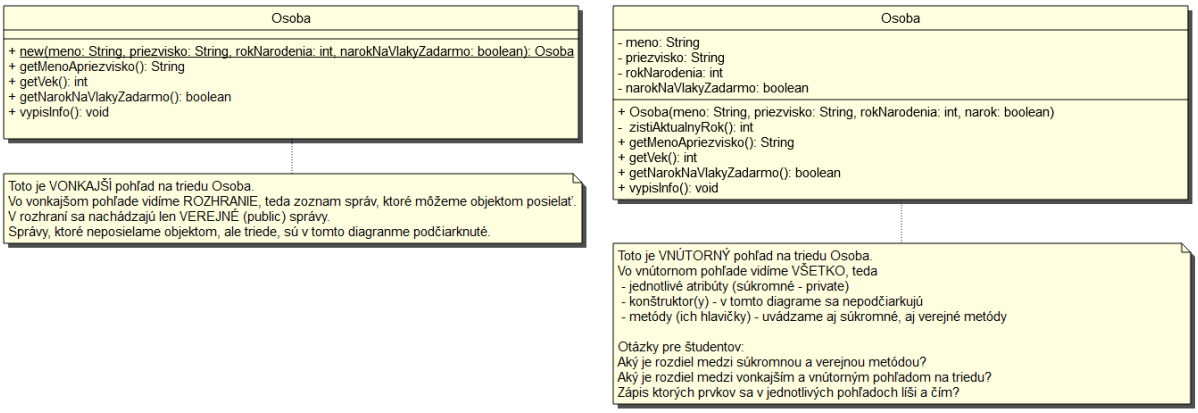
Na zopakovanie: Toto je **vnútorný** pohľad na triedu *Osoba*, kde vidíme všetko (súkromné aj verejné časti triedy).

Prvý príklad je za nami☺

OTÁZKY PRE ŠTUDENTOV:

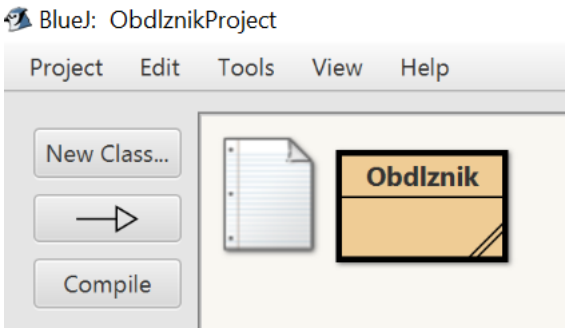
1. Aký je rozdiel medzi vonkajším a vnútorným pohľadom na triedu?
2. Aký je rozdiel medzi súkromnou a verejnou metódou?
3. Zápis ktorých prvkov sa v jednotlivých pohľadoch líši a čím?

Porovnanie **vonkajšieho** a **vnútorného** pohľadu na triedu *Osoba*, aby ste si zapamätali hlavné rozdiely (čo zobrazujeme v ktorom pohľade a akým spôsobom):

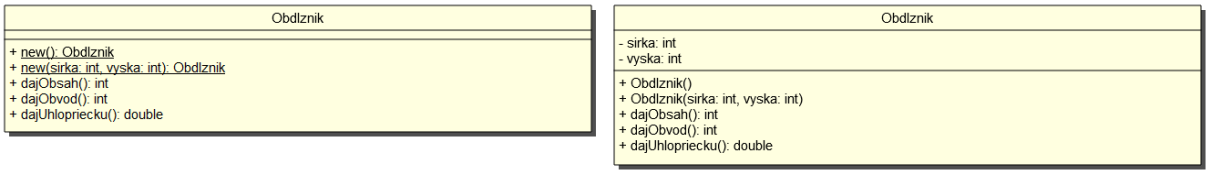


Poznámka: Ak budete pracovať v nástroji UML.FRI, znamienka + a – v diagramoch písať nebudete, nastavia sa automaticky na základe toho, ako nastavíte prístupové práva atribútov a metód. Štandardne sú prednastavené, nič meniť netreba. Pre viac informácií si pozrite popis UML.FRI na konci tohto dokumentu.

Po podrobnom príklade a ukážkach po krokoch je čas na vlastné riešenie.
Otvorte si adresár IpM_cv3_02. Nájdete v ňom podadresár ObdlznikProject, samostatný súbor ObdlznikDiagram.frip2 (výsledný diagram tried nakreslený v nástroji UML.FRI) a obrázok UML_Obdlznik.png (exportovaný obrázok z nástroja UML.FRI). Otvorte si projekt ObdlznikProject v BlueJ. Obsahuje jednu triedu, mali by ste vidieť toto:



Je to prvý príklad z našich spoločných cvičení. Úloha: Pomocou UML vytvorte *class diagram* pre triedu *Obdlznik*. Vytvorte **vonkajší** aj **vnútorný** pohľad. Môžete využiť ľubovoľný grafický (textový) editor, prípadne aj UML.FRI, ak chcete. S pomocou výkladu k predchádzajúcemu príkladu by ste sa mali dopracovať k takémuto výsledku:

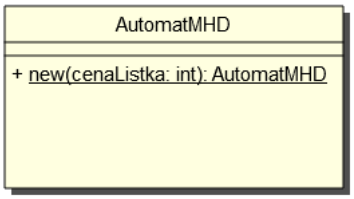


Doteraz sme sa z pohľadu tvorby tried zaoberali ich popisom a grafickým znázornením pomocou UML *class diagramu*. Vieme, ako *class diagram* vyzerá, čo obsahuje. Teda, na základe kódu existujúcich tried vieme vytvoriť UML diagram (vonkajší i vnútorný pohľad).

Class diagram sa však veľmi často využíva už vo fáze návrhu, kedy triedy ešte neexistujú, ale navrhuje sa ich štruktúra (atribúty, metódy, vzťahy medzi nimi). Vyskúšajme si jednoduchý príklad aj my – adresár IpM_cv3_03. Zatiaľ z neho neotvárajte nič.
Úloha: Uvažujme jednoduchý automat na predaj lístkov MHD. Navrhnite triedu *AutomatMHD*. Pomocou UML vytvorte jej vonkajší aj vnútorný pohľad formou *class diagramu*.

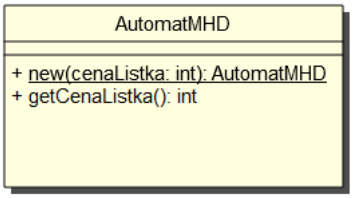
Začnime **vonkajším** pohľadom, teda *správami*, ktoré chceme automatu posielať. Vnútorný pohľad sa nám z vonkajšieho spraví ľahko (atribúty ľahko vydedukujeme zo správ a metódy vlastne už budeme mať).
Aby sme si to trochu zjednodušili, predpokladajme, že automat vydáva len jeden typ lístkov, napríklad základné dvojpásmové (v Žiline aktuálne za 65 centov/kus).
Poďme postupne:

1. Aké správy chceme posielať triede *AutomatMHD*? Nepochybne správu *new*, pomocou ktorej vytvoríme konkrétny objekt, konkrétny automat. Mala by mať nejaké vstupné parametre? Ak vieme, že cena lístka je dnes 65 centov/kus, bude taká aj o rok? Triedy navrhujeme tak, aby boli znovu-použiteľné (*re-use*) bez nutnosti veľkých úprav a zmien v kóde. My nevieme, aká bude cena lístka o rok, ale chceli by sme, aby sme aj o rok mohli vytvárať objekty tejto triedy. Preto je logické, že parametrom tejto správy bude cena lístka (dnes 65 centov, o rok môže byť iná a všetko bude fungovať). Teda, vždy, keď budeme vytvárať objekt (inštanciu triedy *AutomatMHD*), už pri vzniku objektu nadefinujeme cenu, koľko bude stáť v danom automate lístok. Po tomto kroku máme:

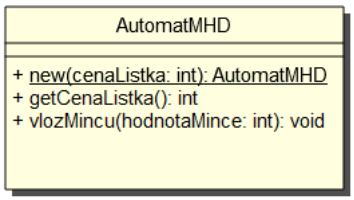


2. Aké správy budeme posielať objektu (konkrétnemu automatu)? Zjednodušene povedané: čo od neho chceme - aké činnosti sú spojené s fungovaním automatu, aké služby nám poskytne?
   Predstavte si, že prídete v neznámom meste k automatu na predaj lístkov MHD. Jeho vlastnosti (*atribúty*) nepoznáte a ani poznať nebudete. Čo vás zaujíma ako prvé? No predsa, koľko stojí lístok, nie? Teda, automat musí byť schopný zobraziť cenu lístka,

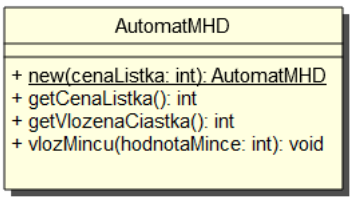
resp. poskytnúť ju navonok. Správa by mohla mať tvar `getCenaListka(): int`. Predpokladajme, že cena lístka bude v centoch a môže mať celočíselnú hodnotu (int).



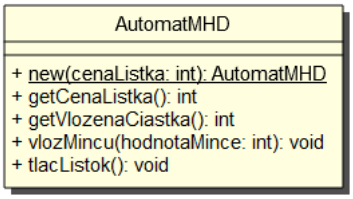
Keď už viete, koľko stojí lístok, chcete si ho kúpiť. Čo robíte? Hádžete do automatu mince. Teda, automat prijíma mince. Správa, ktorá by to realizovala, by mohla mať tvar `vlozMincu(hodnotaMince: int): void`. Metóda nič nevráti, návratový typ môže byť *void*. Aby sme vedeli, akú mincu sme vhodili, parametrom tejto správy bude nominálna hodnota mince.



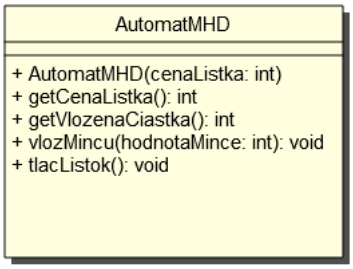
Dokedy budeme vkladať mince? Vždy, keď vložíme mincu, tak na displeji automatu vidíme, koľko sme tam už nahádzali. Inými slovami, automat nám musí poskytnúť informáciu o celkovej vloženej čiastke. Správa `getVlozenaCiastka(): int`.



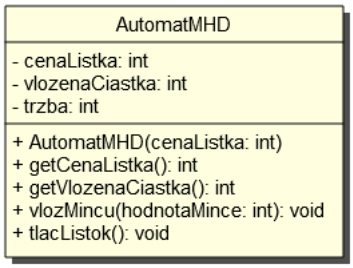
Napokon (keď sme nahádzali dostatočnú hotovosť) nám automat vytlačí cestovný lístok. Správa, ktorú automat dostane, môže mať tvar `tlacListok(): void`.



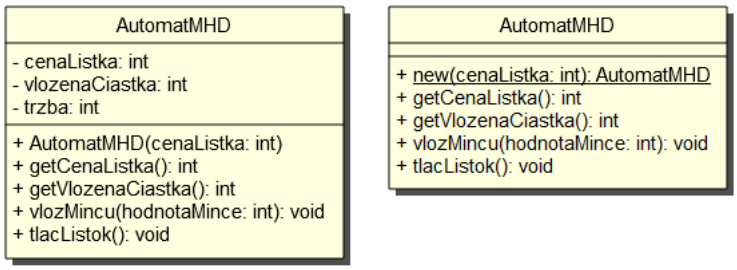
Základná funkcionalita automatu je pokrytá, *class diagram* predstavujúci **vonkajší** pohľad na triedu *AutomatMHD* je hotový.
Ako bude vyzerať automat zvnútra, aké bude mať *atribúty*? Poďme na **vnútorný** pohľad.
Čo vieme prevziať z vonkajšieho pohľadu? *Metódy*. Ako? Metódy skopíruje v takom tvare, v akom sú a zápis konštruktora upravíme na hlavičku metódy (už to nebude správa *new*, ale verejná metóda bez návratového typu s rovnakým názvom ako trieda).



Aké vlastnosti (*atribúty*) má mať automat? Z čoho ich vyčítame? Postupujme opäť logicky. Pozrime sa na to, aké parametre má správa *new*, resp. konštruktor. Ak má konštruktor nejaké parametre, chceme pomocou nich nastaviť vlastnosti objektu, inak by tam tie parametre zrejme nemali zmysel. Teda, ak má konštruktor parameter *cenaListka*, a navyše takúto vlastnosť chceme pomocou správy *getCenaListka()* zobraziť alebo poskytnúť smerom von, potom je logické, že jednou z vlastností automatu bude *cena lístka* typu int.
Ďalšia vlastnosť vyplývajúca zo správ je *vložená čiastka*. Do automatu hádžeme mince rôznych hodnôt - metóda `vlozMincu(hodnotaMince: int): void` a celkovú vloženú čiastku chceme zobraziť (dostať von) metódou `getVlozenaCiastka(): int`.
Keď sa lístok vytlačí, vložená čiastka sa zrejme vynuluje. Aby sme vedeli, koľko sme zarobili, môžeme ešte pridať celočíselný atribút *tržba*, ktorý sa bude aktualizovať pri každom volaní metódy `tlacListok(): void`.
Výsledný diagram pre **vnútorný** pohľad na triedu *AutomatMHD* môže vyzerať takto:



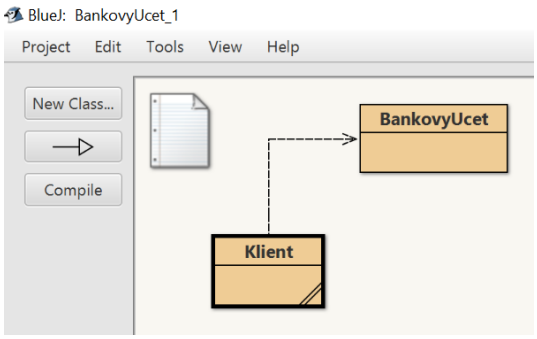
Teraz si opäť porovnajme **vonkajší** a **vnútorný** pohľad na triedu *AutomatMHD*.



Pre záujemcov som podľa týchto UML diagramov naprogramoval triedu *AutomatMHD*, zdrojové kódy nájdete v adresári primitivnyAutomatNaListky. Môžete otestovať.
Úloha pre všetkých na premyslenie: Aké nedostatky má Vami/nami navrhnutá trieda? Upravte svoj návrh tak, aby sa model automatu čo najviac priblížil realite.
Cvičenie č. 3 zamerané na modelovanie tried pomocou UML diagramov zavŕšime dvojicou príkladov s viacerými triedami, aby ste videli aj vzťahy medzi nimi.

Otvorte si adresár IpM_cv3_04. Nájdete v ňom dva príklady (podadresáre BankovyUcet_1 a BankovyUcet_2, UML diagramy nakreslené v nástroji UML.FRI (súbory Ucet_1.frip2 a Ucet_2.frip2) a tiež exporty diagramov do obrázkov (UML_Ucet_1.png a UML_Ucet_2.png).

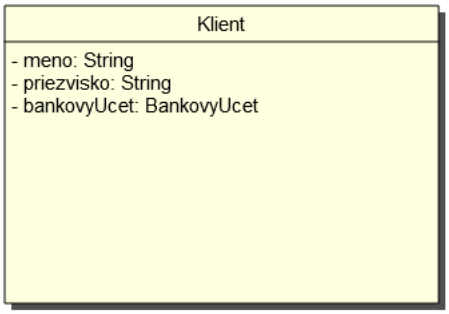
Začnime príkladom BankovyUcet_1. Keď si projekt otvoríte v BlueJ, mali by ste vidieť toto:



Projekt pozostáva z dvoch tried – *Klient* a *BankovyUcet*. My si spravíme UML diagram týchto dvoch tried znázorňujúci **vnútorný** pohľad. Začneme triedou *Klient*. Aké má *atribúty*?

```
public class Klient{
    private String meno;
    private String priezvisko;
    private BankovyUcet bankovyUcet;
```

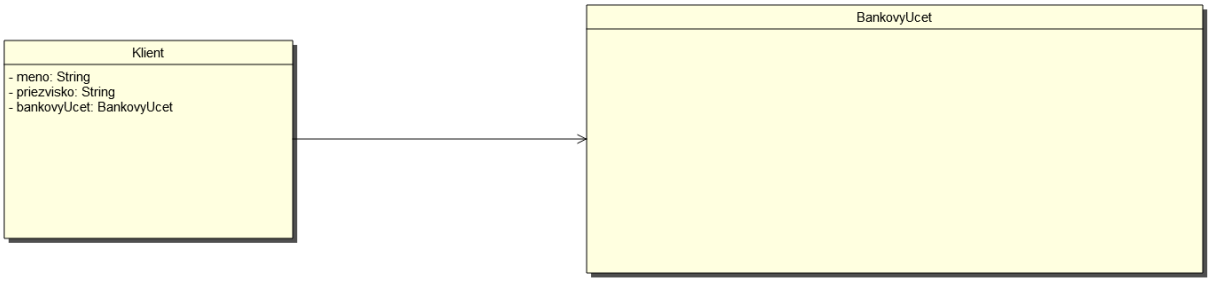
Klient má dva textové *atribúty* typu *String* a jeden objektový typu *BankovyUcet*. Znázornenie v UML diagrame by mohlo vyzerať takto:



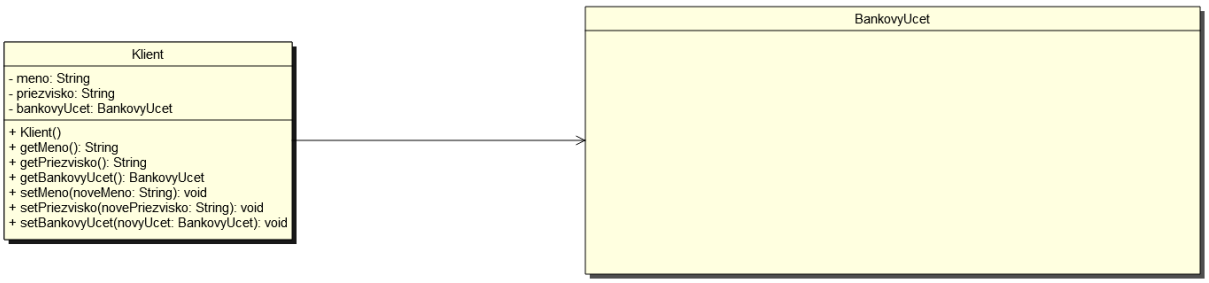
Trieda *BankovyUcet* je naša vlastná (nie je predpripravená, resp. vstavaná v Jave) a preto je už teraz jasné, že medzi triedami *Klient* a *BankovyUcet* bude nejaký vzťah. Aký? Kompozícia alebo asociácia? Kompozíciu by sme použili vtedy, ak by klient a účet mali spoločný životný cyklus, teda, ak by sa klient skladal z účtu, resp. ak by bol účet súčasťou celku, ktorým je klient. To by znamenalo, že súčasne s klientom vzniká aj bankový účet. Keď sa narodí nový človek, ešte nemá účet, teda logicky sa nám kompozícia nehodí. Ak máme k dispozícii kód, môžeme sa pozrieť do konštruktora a tam uvidíme, či sa bankový účet vytvára s klientom alebo sa nastavuje na *null* (zatiaľ neexistuje).

```
public Klient(){
    this.meno = "Marek";
    this.priezvisko = "Kvet";
    this.bankovyUcet = null;
}
```

Aj z kódu je zrejmé, že medzi klientom a účtom bude vzťah asociácie. Ten v UML diagrame naznačíme šípkou.



A môžeme pokračovať ďalej. Do *class diagramu* triedy *Klient* teraz doplníme *metódy*. Prechádzame kódom a všetky hlavičky metód (vrátane konštruktora) prepíšeme do diagramu spôsobom, ktorý sme popísali v predchádzajúcich príkladoch. Dostaneme:



Teraz nám zostáva vyriešiť ešte *class diagram* triedy *BankovyUcet*. Aké má *atribúty*?

```
public class BankovyUcet {
    private String nazovUctu;
    private String PIN;
    private double zostatok;
    private double urokovaSadzba;
    private int pocetTransakcii;
    private double[] transakcie;
```

Na zopakovanie: *String* je údajový typ pre text (sekvenciu znakov, napríklad “Marek”), *int* predstavuje celé čísla a *double* predstavuje reálne čísla. *Double* je v Jave to isté, ako ste mali vo flowgorith typ *real*. Čo je nové v tejto triede, je zápis *double[]*. Je to zápis pre *pole*. Predpokladám, že si aspoň matne spomínate na pole z prvého semestra Informatiky pre manažérov. Ide o to, aby ste v jednej premennej (transakcie) dokázali ukladať niekoľko hodnôt rovnakého typu (*double*). V prípade bankového účtu to budú transakcie, teda čiastky typu *double*, ktoré na účet vkladáte a z účtu vyberáte. Konkrétny príklad by mohol vyzerať napríklad takto (toto by mohlo byť pole typu *double*[6]):

100 eur	-10 eur	-15 eur	20 eur		

Toto ukážkové pole má kapacitu 6 prvkov – kapacita sa udáva pri definícií vnútri zátvoriek [] - a zatiaľ sú využité 4, v ostatných bunkách sú nuly (ale tie nás nezaujímajú). Teda v bankovom účte som vykonal postupne tieto operácie:

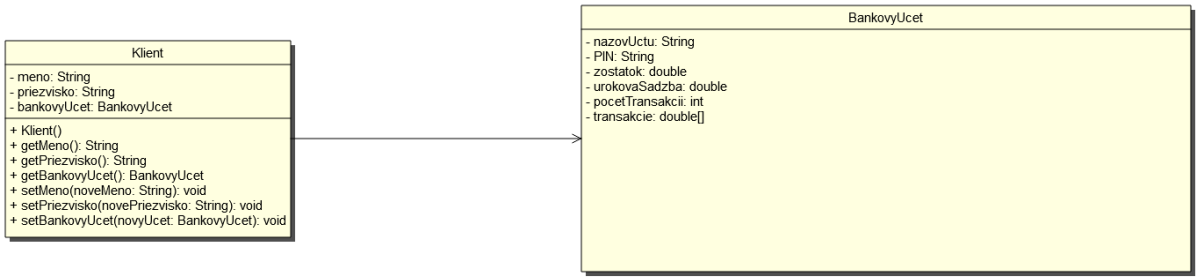
1. Vklad 100 eur
2. Výber 10 eur
3. Výber 15 eur
4. Vklad 20 eur

Pri vytváraní poľa musíme nadefinovať jeho veľkosť (maximálnu kapacitu) a tú nikdy nemôžeme presiahnuť.

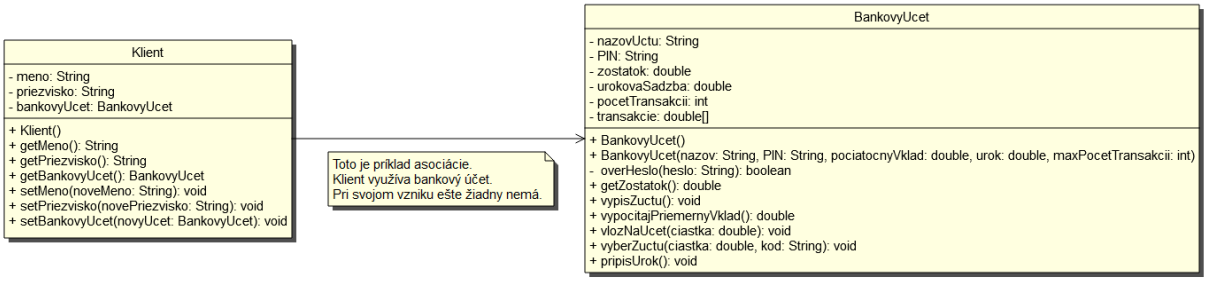
```
public BankovyUcet(String nazov, String PIN, double pociatocnyVklad, double urok, int maxPocetTransakcii) {
    this.nazovUctu = nazov;
    this.PIN = PIN;
    this.zostatok = pociatocnyVklad;
    this.urokovaSadzba = urok;
    this.transakcie = new double[maxPocetTransakcii];
    this.pocetTransakcii = 0;
}
```

Poľu sa špeciálne venovať nebudeme, len som chcel ukázať, že existuje mechanizmus, ako ukladať niekoľko hodnôt naraz. Je to štandard, teda prirodzená súčasť Javy. Kto by mal záujem o podrobnejší výklad poľa, dohodneme sa individuálne...

Vráťme sa ku *class diagramu* triedy *BankovyUcet*. Teraz môžeme vložiť atribúty (už im rozumiete všetkým, aj zápisu []). Dostaneme:



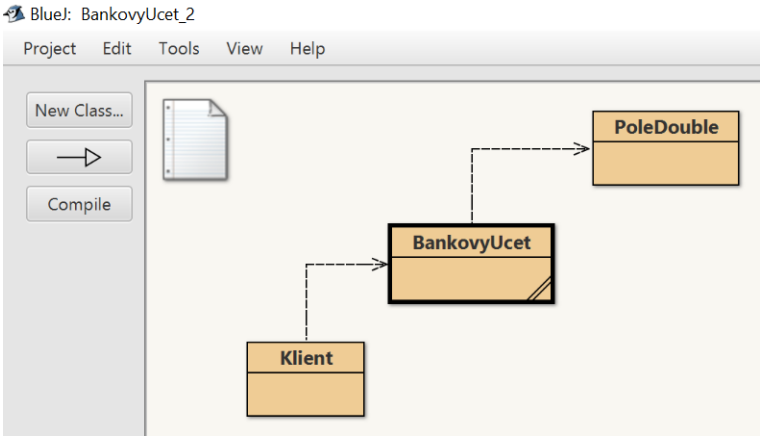
Metódy doplníme štandardným spôsobom. Výsledný UML diagram vyzerá takto:



Na tomto jednoduchom príklade sme ukázali dve nové veci:

1. Ako využiť pole a zapísať ho do diagramu
2. Ako vytvoriť a namodelovať vzťah asociácie medzi dvomi triedami

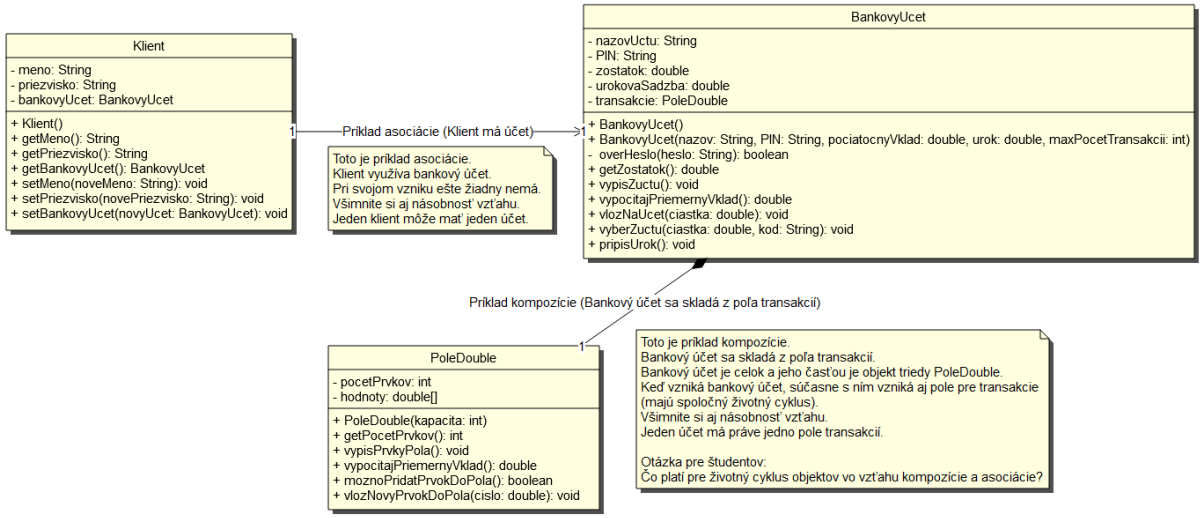
Prípadné otázky k tomuto príkladu Vám ochotne zodpoviem formou konzultácie, kontakty nájdete v úvodnej časti tohto sprievodného dokumentu (email, Skype).
Posledný príklad cvičenia č. 3 bude kombináciou kompozície a asociácie. Otvorte si projekt BankovyUcet_2 v BlueJ.



Klienta a bankový účet riešiť netreba, prevzali sme ich z príkladu BankovyUcet_1. Asociáciu medzi klientom a bankovým účtom sme si už vysvetlili. Poďme sa detailne pozrieť na vzťah medzi bankovým účtom a triedou *PoleDouble*.
Ak by sme v bankovom účte nechceli využívať pole priamo, ale z nejakých dôvodov by sme toto pole potrebovali „vytiahnuť“ do samostatnej triedy, dostali by sme sa do stavu, kedy súčasťou bankového účtu bude objekt triedy *PoleDouble*. Teda, bankový účet sa skladá z poľa transakcií, resp. pole transakcií je súčasťou celku, ktorým je bankový účet. V momente, keď vzniká bankový účet, vzniká aj pole pre jeho transakcie (pozri obrázok). Ďalšou pomôckou pre ľahšiu identifikáciu vzťahu kompozície je otázka, či by objekt triedy *PoleDouble* mohol existovať samostatne, teda, či by takýto objekt dával logický zmysel. Nedával. Objekt triedy *PoleDouble* je vždy súčasťou niečoho väčšieho, teda je časťou celku => kompozícia.

```
public BankovyUcet(String nazov, String PIN, double pociatocnyVklad, double urok, int maxPocetTransakcii) {
    this.nazovUctu = nazov;
    this.PIN = PIN;
    this.zostatok = pociatocnyVklad;
    this.urokovaSadzba = urok;
    this.transakcie = new PoleDouble(maxPocetTransakcii);
}
```

Dostávame sa tak do kompozície. Keďže všetko ostatné vieme, výsledný diagram vyzerá takto:


Klient
- meno: String
- priezvisko: String
- bankovyUcet: BankovyUcet
+ Klient()
+ getMeno(): String
+ getPriezvisko(): String
+ getBankovyUcet(): BankovyUcet
+ setMeno(noveMeno: String): void
+ setPriezvisko(novePriezvisko: String): void
+ setBankovyUcet(novyUcet: BankovyUcet): void
1
Príklad asociácie (Klient má účet)
1
Toto je príklad asociácie.
Klient využíva bankový účet.
Pri svojom vzniku ešte žiadny nemá.
Všimnite si aj násobnosť vzťahu.
Jeden klient môže mať jeden účet.
BankovyUcet
- nazovUctu: String
- PIN: String
- zostatok: double
- urokovaSadzba: double
- transakcie: PoleDouble
+ BankovyUcet()
+ BankovyUcet(nazov: String, PIN: String, pociatocnyVklad: double, urok: double, maxPocetTransakcii: int)
- overHeslo(heslo: String): boolean
+ getZostatok(): double
+ vypisZuctu(): void
+ vypocitajPriemernyVklad(): double
+ vlozNaUcet(ciastka: double): void
+ vyberZuctu(ciastka: double, kod: String): void
+ pripisUrok(): void
Príklad kompozície (Bankový účet sa skladá z poľa transakcií)
1
PoleDouble
- pocetPrvkov: int
- hodnoty: double[]
+ PoleDouble(kapacita: int)
+ getPocetPrvkov(): int
+ vypisPrvkyPola(): void
+ vypocitajPriemernyVklad(): double
+ moznoPridatPrvokDoPola(): boolean
+ vlozNovyPrvokDoPola(cislo: double): void
Toto je príklad kompozície.
Bankový účet sa skladá z poľa transakcií.
Bankový účet je celok a jeho časťou je objekt triedy PoleDouble.
Keď vzniká bankový účet, súčasne s ním vzniká aj pole pre transakcie
(majú spoločný životný cyklus).
Všimnite si aj násobnosť vzťahu.
Jeden účet má práve jedno pole transakcií.
Otázka pre študentov:
Čo platí pre životný cyklus objektov vo vzťahu kompozície a asociácie?