1. Vytvořte projekt Account. V ní třídu Account1.cs, ve které bude namespace Account1 a třída **Account** s datovou složkou **balance** (stav) typu int a metodami **insertInto(amount**), **writeBalance()**, **transferTo(Account, amount)**
Tedy česky: vlož(částka), pisStav(), prevedNa(účet, částka). Metoda insertInto(amount) bude při záporné částce obstarávat i výběr
Konstruktor zatím nepotřebujeme, kdykoliv po vytvoření naplníme datovou složku **stav** metodou **insertInto(amount)**.
Dále přidejte do souboru třídu **TestAccount** , do které umístěte jen metodu Mainx. V ní vytvořte účty u1 a u2. Protože při vytvoření mají nulový stav, tak metodou **insertInto** vložte na účty shodně 100 Kč. Poté převeďte z účtu u1 na u2 50 Kč a vypište stav obou účtů. Nakonec převeďte celý zůstatek účtu u1 na u2 a opět vypište stavy účtů.

1b kap Přetěžování (toto už prezenčně)
Přidejte přetíženou metodu **transferTo(Account)**, tedy jen s jedním parametrem, kde se předpokládá, že se na daný účet převede vše. Tím procvičíme přetížené metody. Čím se liší od překrytých?
???
V druhém kroku tuto přetíženou metodu přepracujte tak, aby volala metodu s oběma parametry. Který způsob volání bude správný: **transferTo(u, balance)** nebo **transferTo(u, u.balance)**? Jaký bude dopad obou variant?

2. Kapitola Skládání tříd (datové složky referenčního typu)
V souboru Account2.cs bude namespace Account2, třída zůstane class Account
Tento krok vytvoříme na základě programu 1, ne 1b. Tedy přetíženou metodu **transferTo(Account)** odstraníme, aby se program příliš neprodlužoval.
První krok: vytvoříme v třídě Account datovou složku **owner** typu řetězce. Použijeme v konstruktoru a v metodě WriteBalance.
Druhý krok: Nespokojíme se s datovou složkou typu řetězec. Tedy chceme o majiteli vypsat všechny dostupné údaje, ne jen jeho jméno. K tomu musíme vytvořit třídu Person. Typ datové složky **owner** změníme z řetězce na Person. Vyměníme i na ostatních dvou místech.
Totéž podrobněji: Vytvoříme (pro přehlednost v tomtéž souboru) třídu **Person** (má datové složky **name** a **age**). A datovou složku tohoto typu přidáme do třídy **Account** a její výpis do metody **writeBalance()**. Tedy instance třídy Account bude znát nejen stav účtu, ale i majitele účtu. Metoda **writeBalance()** pak vypíše kromě zůstatku také jméno majitele (věk vypisovat nemusí). Kompilace proběhne v pořádku, ale při spuštění metody **writeBalance()** program zhavaruje s hlášením **nullReferenceException**. Proč?
???
Proč nehavaruje výpis částky na účtu, když také dosud žádná nebyla stanovena (před voláním metody insertInto(amount))?
???
Takže v dalším kroku musíme vytvořit metodu, která by nastavila majitele. Jak ale zajistit, aby uživatel určitě zavolal tuto metodu dříve, než metodu writeBalance?
Vytvoříme též konstruktor třídy Person.
Překryjte metodu ToString pro účet, zobrazí jméno majitele, tabulátor a stav účtu
V metodě Main budeme nejprve postupovat tak, že vytvoříme objekt majitele, např. o1. Teprve potom objekt účtu u1. Pak totéž pro o2 a u2.

Úkrok stranou: vytvoříme ve třídě TestAccount pod metodou Mainx statickou subrutinu writeBalance2(). Zkopírujeme tam z metody Mainx řádek u1.writeBalance(); u2.writeBalance();
Kompilátor ale ohlásí, že proměnné u1 a u2 zde nejsou známé. Proč?
???
Jak to vyřešíme?
???

3. Doplnění datové složky účtu do třídy osoby
V metodě Main můžeme pomocí u1.writeBalance() vypsat informaci o účtu včetně jména majitele. Případně vypsat jméno majitele účtu u1 přímo pomocí WriteLine. Jak?
???
Můžeme se na to podívat z druhé strany a pro nějakou osobu získat informaci o jejím účtu (o jeho stavu)? Například nějak takto: WriteLine(o1.Account.stav)?
ne
Takže instance třídy **Person** vůbec neví, že má nějaký účet.
Provedeme tedy i opačné provázání. Ve třídě **Person** vytvoříme datovou složku **myAccount** třídy **Account**. Zatím pro jednoduchost předpokládáme, že osoba může mít jen jeden účet. Kdy instanci třídy Person sdělíme, že má účet?
konstruktor
Proto do ?? (viz odpověď na předchozí otázku) doplníme vložení objektu právě vytvářeného účtu do datové složky **myAccount** majitele (tedy osobě sdělíme, že se stala šťastným majitelem nového účtu).
Dosud jsme obvykle v konstruktoru nějaké třídy inicializovali všechny datové složky dané třídy. Budeme v konstruktoru osoby vyplňovat datovou složku Account?
???
Ověříme si, že se majitel skutečně dozvěděl, že má účet a vypíšeme proto v metodě Main pro datovou složku **myAccount** první osoby datovou složku stav

4. Změna datové složky účtu v třídě **Person** na pole účtů (kapitola Pole objektů)
Zatímco účet má jen jednoho majitele, tak majitel může mít více účtů (pole účtů). Proto ve třídě **Person** nahradíme datovou složku **myAccount** polem **myAccounts**. Přidáme také datovou složku **countOfAccounts**. Při tvorbě účtu (v konstruktoru) vložíme vytvářený účet do buňky pole s indexem **countOfAccounts**(jedná se o pole účtů daného majitele). A poté zvýšíme počitadlo **countOfAccounts** daného majitele.

V dřívějším cvičení byla datová složka PocetOsob statická. Je také **countOfAccounts** statická datová složka třídy Person?
???

Druhý krok: Abychom mohli jednoduše vypsat stav všech účtů nějaké osoby, tak do třídy **Person** doplníme metodu **writeAccounts ()**. Tato metoda vypíše všechny účty, které daná osoba má. Výpis jednotlivých účtů pole pomocí WriteLine(mojeUcty[i]) je trochu nesrozumitelný, vypíše se např. Account4.Account. Je to proto, že se jedním příkazem WriteLine vypisuje celý objekt, tedy stav i majitel. Proto se omezíme jen na vypsání položky **balance**, ta již bude čitelná (neboť se jedná o primitivní datový typ, v tomto případě číslo).
A pak zvlášť vypíšeme i majitele. Tato položka však opět není čitelná, protože se jedná o objekt skládající se také ze dvou položek, tedy jména majitele a jeho věku. Takže z položky majitele vypíšeme jen jeho jméno nebo věk. Na jakou konstrukci to vede? Rozumíte těm dvěma tečkám?

Překryjte metodu ToString pro třídu Person, přitom využijte ToString třídy Account. Jednotlivé účty se vypíší pod sebou.

4b: místo pole použijte generickou kolekci List.

V **druhém** kroku předělejte smyčku for na foreach. Nebude už pak potřebná datová složka **countOfAccounts**.

4c: předělejte na kolekci typu ArrayList. Pozor na přetypování při čtení z kolekce ve smyčce for. Proč ve smyčce foreach není vždycky přetypování nutné?

5. Doplnění třídy datum (místo vek)
Abyste více procvičili datové složky referenčního typu, tak přidejte třídu **Date** s datovými složkami **day**, **month** a **year**. Potom ve třídě **Person** použijte datovou složku **dateOfBirth** třídy **Date** místo celočíselné datové složky **age**. Konstruktor třídy Person ponechte **dvoj**parametrický, jen místo parametru age bude **dateOfBirth**. Upravte i

třídu **Account,** v metodě **writeBalance** se bude i nadále vypisovat věk, ale bude se muset vypočítat např. jako rozdíl aktuálního letopočtu a roku data narození majitele.
DateTime.Today.Year
Kolik teček pak bude ve výpisu například roku narození v metodě **writeAccounts()** ve třídě **Person**? Jakým způsobem to číst?
???

Jak v hlavním programu vypíšete rok narození majitele účtu1?

???

Jak tomu rozumíte?

Druhý krok: Reference d1 na instanci třídy datum vzniká zbytečně, použije se jen při tvorbě osoby. Proto u druhé osoby vytvoříme instanci datumu přímo v závorce konstruktoru osoby.

Instance datumu bude tedy mimo konstruktor neznámá. A půjdeme ještě dál: zrušíme vytvoření osoby o2 a také ji vytvoříme pomocí new až při tvorbě účtu, nebude jí přiřazena proměnná (bude tedy anonymní).

I přesto můžeme datové složky této osoby vypsat, ale nepřímo, jako majitele účtu: u1.owner.dateOfBirth.day ap. (writeInfo tu neni, to byla soucast preskocene kapitoly)

6. **Třetí** krok: vytvoříme přetížený konstruktor osoby, který bude mít **čtyři** parametry. Kromě jména i celá čísla den, měsíc a rok.

Ověříme vytvořením osoby v metodě Main.

Pozor: když v konstruktoru třídy Person dáme dateOfBirth.day=d atd., aniž by byl new Date, tak to havaruje až za běhu.

Čtvrtý krok: Tento přetížený konstruktor bude volat předchozí konstruktor (ten se dvěma parametry). Proveďte to tak, ať nemusíte před voláním konstruktoru vytvářet instanci třídy Date.

7. Doplnění zadávání z klávesnice

V dalších třech krocích chceme procvičit práci s výjimkami. Nemůžeme však vyjít z bodu 6, protože obsahuje tři třídy, což by bylo nepřehledné. Vrátíme se tedy k bodu 1 (takže rezignujeme na možnost zadávat majitele účtu). Zkopírujte soubor Account1.cs a přejmenujte na Account7.cs. Doplníme do metody Main zadání z klávesnice částky, kterou pak metodou insertInto(amount) vložíme na tento účet. Odstraníme metodu transferTo() s jedním parametrem.

8. Ošetření výjimky dělení nulou (kap Výjimky)

Vytvořte metodu writeBalanceInDolars(int kurz). Doplňte volání této metody na konec metody Main. Kurz při volání zadáme číselně, např. writeBalanceInDolars(20). Pokud je v tu chvíli stav účtu např. 200 Kč, tak se vypíše 10 dolarů.

V druhém kroku zjistěte, co se stane, když omylem metodu zavoláme writeBalanceInDolars (0)? Doplňte do metody writeBalanceInDolars zachycení příslušné výjimky.

Pozn.: parametr kurz musí být definován jako celočíselný. To je sice poněkud nesmyslné, ale jinak bychom nemohli demonstrovat výjimku dělení nulou. Pokud by se dělilo nulou jako reálným číslem, tak by k havárii nedošlo, výsledek by se vypsal jako otazník (ve starších VS se vypisovalo +nekonečno). Ověřte.

V třetím kroku ověřte hierarchii výjimek, zachyťte i výjimku nadřazenou výjimce DivideByZeroException. Která to je? A která je ještě výše?

Aby se zachytila, musíme zakomentovat catch DivideByZeroException

V jakém pořadí se výjimky mají zachytit? Jaké hlášení dostaneme, když bude pořadí opačné?

Vyvolání výjimky: Ve čtvrtém kroku přidáme metodu writeBalanceInDollarsDouble. Parametr „kurz“ této metody bude typu double. Protože v tom případě nedojde při dělení k vyvolání výjimky DivideByZeroException, vyvoláme ji sami (protože dělení reálnou nulou také nemůžeme označit za vyhovující)

9. kap Výjimky/Vyvolání výjimky

Založíme na bodě 7 (tedy dělení nulou ani writeBalanceInDollars nebudeme dále používat)

Program při záporném zůstatku sice nehavaruje, ale typ našeho účtu to zakazuje. Proto v metodě **insertInto(amount)** otestujeme, zda je dost peněz. Pokud ne, sami vyvoláme výjimku a přitom jí předáme řetězec vysvětlující problém (že pro **výběr** není dost peněz). Použijeme k tomu některý vhodný typ výjimky, např. ArgumentOutOfRangeException. Zadání nevhodné částky pak program nepovolí, zhavaruje. A doplnit hlášení při nedostatku peněz, že se zobrazí, kolik peněz tam je.

Proto v **druhém** kroku doplníme do metody blok try – catch. Program pak již pracuje v pořádku, při nedostatku peněz vypíše hlášení (tedy nezhavaruje), k převodu peněz nedojde a metoda končí.

V **třetím** kroku totéž proveďte s metodou **transferTo()**. Text bude: „pro **převod** není dost peněz“.

Ve **čtvrtém** kroku vyhoďte i výjimky pro případ, že by uživatel zadal zápornou částku (text: Nesmíš vysávat cizí účty) nebo převáděl 0 Kč. Nebo posílal peníze sám sobě (tedy zdrojový i cílový účet stejný).

10. Zachycení výjimky ve volající metodě (kap. Výjimky/ Vyvolání výjimky a vyhození do volající metodě)

V předchozím bodu sice výjimka nezpůsobila havárii programu, ale po vypsání chyby program skončil. Metoda sice neprovedla nepovolenou transakci, ale nedala nám možnost zadat jinou částku. Výjimku totiž zachytáváme v metodě insertInto(amount), zatímco z klávesnice čteme v metodě Main(). Proto výjimku nebudeme zpracovávat v metodě insertInto(amount). Tím způsobíme „probublání“ výjimky nahoru do volající metody, tedy do metody Main()

Tedy jen se pomocí CTRL+X a CTRL+V přesune try{ do metody Main nad volání u1.insertInto() a pak se přesune }catch(...){...} pod volání u1.insertInto().

V dalším kroku doplňte do sekce catch vypsání vlastnosti StackTrace.

11. Zajištění opakovaného zadávání

Abychom měli z přesunutí bloku try-catch do hlavního programu nějaký užitek, zajistíme, aby ve smyčce do-while byl uživatel tak dlouho vyzýván k zadání jiné částky pro **vložení** peněz, dokud se nepodvolí. Smyčka bude nekonečná, bude se z ní vyskakovat pomocí break.

11b

Druhý krok: Nyní smyčka pro zadání částky pro **převod** (t.j. transferTo): tuto smyčku provedeme jiným způsobem. Nebudeme vyskakovat pomocí break, místo toho si vytvoříme proměnnou ok typu boolean, která se bude ve smyčce testovat (smyčka skončí, když bude ok==true). Kde se bude tato proměnná nastavovat na true a kde na false?
???

Třetí krok: Co se stane při převodu na jiný účet, když má někdo na svém účtu nulu?

12. UcetVlastni: ekvivalent bodů 9. a 11 s použitím vlastní třídy výjimky. Tedy nepoužijeme zabudovanou třídu OutOfRangeException, ale vytvoříme si vlastní třídu **MaloPenezException**.

UcetVlastni9. Zatím má prázdné tělo. Proč nemůžeme při vyhození výjimky zadat v závorce vysvětlující text?

???

Proto výjimku vyvoláváme s prázdnou závorkou, nemůžeme jí tedy při vyvolání předat vysvětlující text, zda k výjimce došlo při vložení nebo převodu peněz, případně k jaké ze tří chyb při převodu peněz došlo. Proto tento text píšeme až při zachycení výjimky. Dokud výjimku zachytáváme v metodě, kde vzniká, tak není problém. Neboť je jasné, zda k výjimce došlo při vložení nebo převodu.

Ale úplně ideální to není, v rámci převodu nezjistíme, který ze tří stavů nastal. Proto musíme vytvořit ještě VyberZCizihoUctuException a PrevodNulyException. Srovnejte výpis e, e.Message, e.StackTrace a nic

13. UcetVlastni11 (UcetVlastni10 nebudeme vytvářet) Vycházet budeme z programu Account11
Pro jednoduchost zde budeme vyhazovat jen výjimku, kdy je málo peněz (pro převod nebo pro výběr). Tedy už nebudeme pokračovat s výjimkami VyberZCizihoUctuException a PrevodNulyException.
Při zachytávání výjimky v hlavním programu však už nemáme informaci, z které metody byla výjimka vyhozena, proto překročení částky při výběru a převodu nerozlišíme. Proto bychom potřebovali, aby informaci o výběru nebo převodu peněz si s sebou výjimka nesla již od svého vzniku.

Tedy potřebujeme konstruktor s řetězcovým parametrem. Pak budeme moci vlastní výjimku vyvolat stejně jako v předchozích krocích výjimku OutOfRangeException.