

1. Základ bez konstruktoru a dědičnosti (kap. První OOP, str 2)

Vytvořte solution Csharp_OOP_cv a v něm projekt První typu Console Application.

Automaticky se v něm vytvoří třída Program.cs s metodou Main(). Vytvořte v projektu třídu Prvni1.cs, v ní veřejnou statickou metodu Mainx().

Zavolejte metodu Mainx z metody Main třídy Program.cs. Ověřte, zda program chodí (i když dosud nic nedělá)

Do souboru Prvni1.cs doplňte třídy Student, Accountant a Teacher (tedy všechny tři budou v jednom souboru s třídou Prvni). Všechny tři budou mít datovou složku age, Student navíc scholarship, Accountant a Teacher místo toho salary. A Teacher navíc třetí datovou složku: teachingTime (počet úvazkových hodin). Všechny datové složky budou celočíselné. Všechny třídy budou obsahovat metodu writeInfo(), která vypíše hodnotu všech datových složek.

Do metody Mainx() umístěte vytvoření po jedné instanci z tříd Student, Accountant i Teacher (nazvěte je např. s1, a1, t1) včetně naplnění datových složek nějakými hodnotami. Potom datové složky každé instance vypíšte. V tomto bodě ještě nepoužívejte konstruktory ani dědičnost a dejte si pozor, abyste nedělali třídy jako vnitřní!

Kdo má - 1 do Krok 1 :-)

2. Konstruktor (kap. str. 5)

Program budeme dále upravovat. Z výukových důvodů je ale vhodné, abychom původní verzi pro srovnání ponechali. Proto vytvořme novou třídu, program do ní zkopírujme a namespace z Prvni1 přejmenujme na Prvni2. Podobně budeme pak postupovat i po dalších úpravách. Postup:

a) Add/New/Code file, nazvat jej Prvni2.cs

b) CTRL+A vybrat celý předchozí program a pomocí

CTRL+C zkopírovat do schránky

CTRL+A v novém programu, tím se celý vybere

CTRL+V vložit do nového souboru

c) Namespace přejmenovat na Prvni2 (jinak by bylo hlášení o ambiguity) a můžeme stejně přejmenovat i třídu.

d) Přidat do Program.cs volání Mainx() v nové třídě (v namespace Prvni1), zakomentovat volání předchozí. Protože je metoda Main v jiném namespace než Mainx, je nutné uvést celou cestu, tedy jmenný prostor.třída.metoda. Ověřte spuštěním

Pozn.: proč musíme přejmenovat namespace? Protože v každém kroku programu budeme mít vždy třídy Student, Accountant a Teacher. Aby nedošlo k nejednoznačnosti, tak budou jednou v jmenném prostoru Prvni1, pak v Prvni2 atd. Název třídy obsahující metodu Main se měnit může (class Prvni1, Prvni2 atd.) ale nemusí (tedy stále class Prvni).

A nyní zadání: Představme si, že každá třída má ve skutečnosti desítky datových složek.

Tedy by vytvoření každé instance zabralo desítky řádků. Proto program zkrátíme použitím konstruktoru. Vytvořte v každé třídě konstruktor inicializující všechny potřebné datové složky (u třídy Teacher tedy tři datové složky). Upravte pak v metodě Mainx() tvorbu instancí tak, aby se použily konstruktory. Tvorba každé instance je pak na jeden řádek. Úspora se samozřejmě projeví až při větším počtu vytvořených instancí.

3. Přidání dědičnosti (kap. Dědičnost)

Pozor: ! Nedělat úkoly dopředu, v nových třídách zatím nebudeme vytvářet konstruktor ani metodu writelInfo(). Tedy třídy Person i Employee budou mít jen jeden řádek. A nevolat ještě :base(), tedy nevolat nadřazený konstruktor z podřízeného, to je až v bodě 5

Vidíme, že v programu je opakovaně deklarována datová složka age, také metoda writelInfo() a konstruktory jsou z větší části stejné. Zvláště při větším množství tříd by byl program zbytečně dlouhý. Proto provedeme analýzu, zda bychom mezi objekty nenašli znaky dědičnosti (tedy zda nemají třídy některé datové složky společné). Vytvoříme tedy třídu Person, která bude obsahovat datovou složku age, a ostatní tři třídy z ní budou dědit (doplňte za dvojtečku). Odstraňte z tříd Student, Accountant a Teacher datovou složku age, kterou již nepotřebují, neboť ji podědily.

Nejprve si to vyzkoušíme pro třídu Student. Pokud se program spustí v pořádku, provedeme tytéž změny i pro ostatní dvě třídy.

!! Takto budeme postupovat i v dalších bodech: nejprve změnu odzkoušíme na třídě Student a program spustíme. Teprve bude-li vše v pořádku, upravíme i další třídy!!

2. krok: Pokračujeme v analýze dále a zjistíme, že třída Accountant a Teacher mají společnou datovou složku salary. Takže vytvoříme jim nadřazenou třídu Employee, který bude dědit z třídy Person a navíc bude přidávat datovou složku salary. Takže Student a Employee budou dědit z třídy Person. Teacher a Accountant pak budou v hierarchii ještě níže, budou dědit z třídy Employee. Odstraňte z tříd Accountant a Teacher datovou složku salary, kterou již nepotřebují. Pozor: v nových třídách zatím nevytváříme konstruktor ani metodu writelInfo().

4. Přidání konstruktoru do nadřazené třídy (kap. Dědičnost/Dědičnost konstruktorů)

Pozor! Nedělat úkoly dopředu, tedy **nevolat ještě :base()**, tedy nevolat nadřazený konstruktor z podřízeného, to je až v bodě 5

Konstruktory podřízených tříd nejsou naprogramovány efektivně. Zbytečně všechny nastavují například age. Ve třídě Person proto vytvoříme konstruktor nastavující age, ve třídě Employee bude nastavovat salary (jiné konstruktory neměníme). Zkusíme program zkompileovat, objeví se hlášení, že chybí bezparametrový konstruktor **Person()** a **Employee()**. Proč předtím nechyběl? Vždyť nebyl ani předtím! A navíc stejně ani nikde není volán!

*Řešení: Při tvorbě objektu podřízené třídy se vždy nejprve volá konstruktor třídy nadřazené. Není-li volán explicitně konstruktor nadřazené třídy s parametrem, pak se volá konstruktor bez parametru, tedy jako by na začátku konstruktoru bylo :base(). Ten je standardně k dispozici, pokud nadřazená třída nemá nějaký jiný konstruktor. Ve třídě **Person** byl vytvořen konstruktor s parametrem age, proto implicitní konstruktor už není k dispozici a musíme jej napsat sami.*

5. Volání nadřazeného konstruktoru (:base) z podřízeného (tataž kap.) (5b: přetěžování)

V minulém kroku jsme vytvořili konstruktor Osoby s parametrem, ale ještě není odnikud volán.

Upravíme tedy konstruktor ve třídě **Student**, bude nejprve volán bezparametrický nadřazený konstruktor **:base()**. Nic jiného neměníme. Program dále funguje. To je důkaz, že opravdu je implicitně volán nejprve bezparametrický konstruktor nadřazené třídy.

Volání bezparametrického nadřazeného konstruktoru nám ale nepřináší žádnou výhodu. Proto jej změníme na volání konstruktoru s parametrem: **:base(age)**, u učitele a ekonomky **:base(age, salary)** a potom bude přiřazen už jen datová složka, který je proti nadřazené třídě navíc (pro kontrolu: **Accountant** nemá navíc žádnou datovou složku).

Ověřte, že když už teď z tříd Student, Accountant a Teacher voláme explicitně parametrický konstruktor nadřazené třídy, tak již bezparametrický konstruktor třídy Employee ani Person nepotřebujeme (zkusíme jej zakomentovat). Pak jej odkomentujeme, budeme jej potřebovat v bodě 8

5b: (kap. Přetěžování) zatím máme ve Studentovi konstruktor plnící všechny datové složky. Přidejte ještě konstruktor s parametrem jen věk. Potom zkuste přidat konstruktor s parametrem jen scholarship. Zjistíte, že to nejde, proč?

Protože mají stejnou signaturu (kdyby např. scholarship bylo reálné číslo, pak by to šlo).

Tak např. konstruktor na scholarship zrušte a nechte jen konstruktor plnící věk. V metodě Main pak vytvořte studenta s2, pro kterého tento konstruktor použijete. V writeInfo zjistíte, že scholarship má nulové. Můžeme samozřejmě po vytvoření s2 naplnit scholarship přiřazovacím příkazem. V dalším bodě tento konstruktor i s2 zrušíme.

6. Přidání metody writeInfo() do nadřazené třídy (kap. Překrývání - úvod)

Stále jsou neefektivně naprogramovány metody writeInfo. V případě, že by třídy měly desítky shodných datových složek, tak by měly zbytečně desítky shodných řádků. Hierarchie by se tedy dala použít i pro metodu **writeInfo()**. Nadřazené třídy **Person** a **Employee** by měly zajistit výpis datových složek, které jsou v nich deklarovány (age, resp. salary). Takže ve třídě **Student** pak stačí, když metoda **writeInfo()** zavolá stejnojmennou metodu z nadřazené třídy (**base.writeInfo()**) a pak sama navíc vypíše už jen scholarship. Obdobně tato metoda ve třídě **Employee** navíc vypíše salary a v třídě **Teacher** úvazkové hodiny. Ověřte, že příkaz **base.writeInfo()** nemusí být v metodě první.

Co by se stalo, kdybychom prefix **base** neuvedli? Kompilátoru by to nevadilo, při běhu by se však program „zauzloval“, ohlásil by `stackOverflowError`. Proč?

*Protože pak by se nevolala metoda **writeInfo** nadřazené třídy, ale vlastní třídy. Tedy by metoda **writeInfo()** volala samu sebe. A to opakovaně (cca tisíc krát), dokud by zásobník nepřetekl návratovými adresami.*

7. Demonstrace rozdílu ve volání překryté a nepřekryté metody (kap. Překrývání/Mod. virtual, override, new)

Zkopírujte ve třídě Person metodu writeInfo(). Kopii nazvěte jen info(). V metodě writeInfo() třídy Student tuto metodu zavolejte (takže se vlastně dvakrát volá totéž). Je nutno použít při volání také base.info() nebo stačí jen info()? Připomínám, že se ze Studenta volají dvě metody v třídě Person (tedy v nadřazené, básové třídě). U metody writeInfo() se base musí použít (jinak volá stejnojmennou metodu v třídě Student). Musí se base použít i u info()?

Ne, ve třídě Student metoda info() není, takže když ji v této třídě nenajde, hledá ji běhové prostředí automaticky v nadřazené třídě.

V dalších etapách již metodu info() nebudeme používat.

Druhý krok: Podíváme-li se na seznam Warning, vidíme, že kompilátor si není jistý, zda k překrytí metody writeInfo došlo úmyslně. Proto přidáme před překrývanou metodu ve třídě Person klíčové slovo virtual. Tím dáme najevo, že s překrytím počítáme. U překrývajících metod přidáme klíčové slovo override, tím dáme najevo, že překrýváme úmyslně.

8. Doplnění statické proměnné s počtem instancí (kap. Modifikátor Static)

Třidu Person doplníme o statickou datovou složku count (tedy počet). V konstruktoru ji inkrementujeme.

Doplníme její výpis do metody writeInfo() v třídě Person.

Doplňte v třídě Student konstruktor bez parametru (snippet ctor-TAB-TAB), musí existovat i bezparametrický konstruktor třídy Person. V hlavním programu vytvořte dalšího studenta, tentokrát bez parametrů. Také pro něj spusťte výpis writeInfo(). Je údaj o počtu osob správně?

*Nechybí jedna osoba? Pokud ano, pak se podívejte, zda jste nezařadili ve třídě **Person** příkaz **count++** jen do konstruktoru s parametrem age. Bezparametrický konstruktor jste možná ponechali bez povšimnutí.*

9. Změna datové složky na privátní (kap. Zapouzdření)

Předělte datovou složku `age` třídy `Person` na privátní a statickou datovou složku `count` na `protected`. Pokud se s touto proměnnou pracuje jen v třídě `Person`, pak se nic neděje. Jakmile však doplníme výpis této proměnné do jiné třídy, tak třída nepůjde skompilovat. Ověříme si to snahou vypsát hodnotu těchto datových složek v metodě `writeInfo()` ve třídě `Employee` (dědí z třídy `Person`) a v metodě `Main`, která je ve třídě `Prvni9`, která nedědí z třídy `Person`. Jak to bude vypadat s kompilací? Půjde něco zkompilovat?

Jen datová složka `count`, která je `protected`, ve třídě `Employee`.

Musíme tedy vytvořit přístupové metody `getCount()`, `getAge()` a `setAge()` v třídě `Person`, která bude mít přístup `public`. Budou to metody instance nebo statické metody?

`getCount()` bude statická

Pozor: použijeme běžné gettery a settery `getCount()`, `getAge()` a `setAge()`. Nebudeme vytvářet Vlastnosti (Properties) ze stejnojmenné (následující) kapitoly

Pak použijeme volání těchto metod v místech, která nešla skompilovat.

Druhý krok (kap. Vlastnosti) Předělte datovou složku `teachingTime` (ve třídě `Teacher`) na privátní. Vytvořte k ní vlastnost `TeachingTime`. Hlídejte například, že nesmí být větší než 40. Ověřte pomocí `WriteLine` v metodě `Main`.

10. Změna třídy `Person` na abstraktní (kap. Třída a metoda typu `abstract`)

Nezačínat prohlášením třídy za abstraktní!!

Vyjdeme z 9a, tedy nebudeme používat vlastnosti (aby se program zjednodušil).

Nasimulujeme situaci, kdy programátor zapomene ve třídě `Student` přepsat metodu `writeInfo()`. Zakomentujeme metodu `writeInfo()` ve třídě `Student` (v ostatních ponechte). Program se pak začne chovat podivně: metoda `writeInfo()` vypíše všem všechny údaje, studentovi ale jen věk (scholarship ne). Proč?

Protože když `Student` danou metodu nemá, zkouší se zavolat stejnojmenná metoda z nadřazené třídy.

Je nutno zajistit, aby překladač chránil před možnou sklerózou programátora. Proto prohlásíme metodu `writeInfo()` ve třídě `Person` za abstraktní. To zároveň znamená, že zakomentujeme její tělo včetně složených závorek (hlavičku ponecháme). Kompilátor nás donutí, abychom pak za abstraktní prohlásili i celou třídu `Person`. Nemůžeme potom ale bohužel v podřazených třídách tuto metodu volat, zakomentujeme tedy řádek `base.writeInfo()` a místo toho zajistíme výpis věku v každé zvlášť (tedy místo volání `base.writeInfo()` napíšeme `Console.WriteLine(age)`). Při kompilaci pak zjistíme, že tentokrát překladač na absenci metody `writeInfo()` ve třídě `Student` upozorní (a o to nám šlo). Text kompilační chyby: `Student doesn't implement inherited abstract member Person.writeInfo`.

Pozn.: užitečné bude prohlásit za abstraktní i třídu `Employee`, protože stejně nechceme povolit vytváření instancí této třídy (metodu `writeInfo()` ale na abstraktní nepředěláváme). Srovnejte: třídu `Employee` měníme v abstraktní dobrovolně, zatímco u třídy `Person` jsme k tomu byli nuceni, protože obsahuje abstraktní metodu. Platí to i naopak, tedy musí být v abstraktní třídě všechny metody také abstraktní?

ne

11. Použití metody `ToString()`

Vypište v hlavním programu informace o objektu studenta způsobem, jako by to byla proměnná primitivního typu, např. **`Console.WriteLine(s1)`**. V jakém tvaru se informace vypíše?

Prvni11.Student

Na dalším řádku napište: **`Console.WriteLine(s1.ToString())`** vypíše se totéž, čímž je dokázáno, že **`WriteLine`** volá **`ToString()`**.

Jelikož tato metoda není definovaná v našem programu, tak se zřejmě dědí z třídy `Object`.

Vytvořte ve třídě **`Person`** překrytí metody **`ToString()`** třídy **`Object`**. Nejprve neuvedeme klíčové slovo `override`. Zjistíme, že tentokrát oba řádky `WriteLine` stejné nejsou. Řádek `WriteLine(s1)` tuto metodu nevolá, píše postaru `Prvni12.Student`. Proč?

Kompilátor chybu sice nezahlásí (jen varování), ale nedojde k překrytí, takže WriteLine použije nadřazený ToString() třídy Object. Varování zní: 'Prvni12.Person.ToString()' hides inherited member 'Object.ToString()'. To make the current member override that implementation, add the override keyword. Otherwise add the new keyword.

*Bez override není tato metoda ToString příkazem Write/WriteLine volána. Protože bez modifikátoru to kompilátor přeloží, jako by byl uveden modifikátor **new**. Tedy jedná se o úplně jinou, jen náhodou stejnojmennou metodu. K takové se WriteLine nezná*

Co se stane, když napíšeme malé „t“, tedy toString (a vysvětlit)

Tentokrát kompilátor hlásí chybu, že nebyla nalezena žádná vhodná metoda, kterou tímto překrýváme. Pokud pak odstraníme „override“, tak se to chová stejně jako předchozí případ (tedy Write tuto metodu ignoruje, použije nadřazený ToString() třídy Object.

2. krok: chceme-li vracet řetězec pěkně formátovaný, použijeme String.Format. Vyzkoušíme ve třídě Student. Povšimněte si, že metoda Format je statická, metoda ToString instanční.

return base.ToString() + String.Format(" scholarship. {0}", scholarship)

12. Oddělení třídy do samostatných souborů

Oddělte jednotlivé třídy do samostatných souborů (a nic neměňte). A pak si ještě vyzkoušejte změnit ručně v Prvni12.cs jméno namespace např na Prvni13. Co se stane? A pak to zkuste provést pomocí Refactor/Rename.