(Ne)škodný úvod do OOP v jazyce C#

Lukáš Kotek <lukaskotek@gmail.com>

Tento dokument nemá ambici v žádném slova smyslu suplovat technickou dokumentaci, odborné publikace nebo učebnice, které se zabývají programováním a programovacím jazykem C#. Rovněž není jeho cílem poskytovat dokonale exaktní formulace a definice pojmů ze zmíněné oblasti. Naopak cílí na uživatele (a zde konkrétně na studenty středních škol), kteří do objektového programování zatím pouze jemně vplouvají a potřebují proto nenásilnou formou osvětlit základní pojmy a principy, které s objektovým programováním souvisí. Prosím nazírejte na něj s vědomím výše zmíněného.

Chapter 1. Co je to objektově orientované programování

1.1. Proč použít objektové programování

Objektově orientované programování (OOP) je o zcela **jiném přístupu k programování**, než který dříve byl (a samozřejmě i v mnoha případech a z dobrého důvodu dosud je) používán. Jakým způsobem jsme se dívali na program dříve? Jako na sled po sobě jdoucích pokynů. Čili:

1. Krok první: udělej to (třeba načti hodnotu do proměnné);
2. Krok druhý: udělej ono (umocni prve načtenou proměnnou na druhou);
3. Krok třetí: pokud platí A, proveď B (pokud je číslo nenulové, použij ho jako dělitel).
4. Následující krok: …

A podobně bychom mohli pokračovat, je to zjevné?

Algoritmus jsme popsali vývojovým diagramem, který pro nás byl přesným návodem pro to, co kdy a jak udělat. Program pak pro nás pak byl **strukturovaným přepisem tohoto návodu** do konkrétního programovacího jazyka (s jemu poplatnými příkazy a specifickou syntaxí).

Pojďme se podívat na objektově orientované programování (je to dlouhý pojem, pojďme si vystačit se zkratkou OOP, platí?). Předně, OOP vůbec neříká, že to, co je v prvním odstavci není pravda. Algoritmus bude vždy posloupností kroků. OOP říká, že se mění náš pohled na program jako takový. Hlavní pro nás už není postup řešení (aneb co a jak jde za sebou), ale **to s čím vlastně pracujeme** (a o co se vlastně jedná, z čeho se to skládá, jak se to chová, jak to popíšeme?).

Všimli jste si tajemného „ono“? To ono, tajemná věc, o které mluvíme, se nazývá objekt.

1.2. Něco teorie k základní terminologii

1.2.1. Objekt

Máme tu tedy tajemný pojem objekt. Ale on vlastně vůbec není tajemný. Objekt je prvek, kterým se snažíme popsat něco z reálného světa. Co nás napadá? Vezměme těžkou klasiku. Člověka, ale ne ledajakého člověka, vezměme **konkrétního člověka**, spolužáka, kamaráda, např. Igora Nováka (budeme jeho služeb v průběhu textu využívat i dále). Co bychom o něm mohli říct? Třeba, že je nějak vysoký, umí nějaké věci; dá se čekat, že když spadne ze schodů, bude na to nějak reagovat.

Všimněte si toho hlavního:

1. Nás nezajímá jen tak ledajaký člověk (těch je spousta), nás zajímá Igor Novák s konkrétními vlastnostmi, schopnosti a dovednostmi. Mohli bychom univerzálně popsat obecného člověka s přesně definovanými vlastnostmi a schopnostmi? Sotva, každý je něčím specifický. Ale Igor Novák? Ten je náš jeden konkrétní objekt.

2. Nás nezajímají jen samotné údaje tohoto člověka. Nás nezajímá čistě hodnota cholesterolu, ani výška, ani váha. Ne. Pro nás je zásadní, že je to Igorova hodnota cholesterolu, Igorova výška a Igorova váha. Nejsou to samostatně stojící prvky, pevně se váží k Igorovi, popisují ho, jsou s ním spjaté. Igor je objekt, je jedinečný.

Jeli-to jasné, pojďme se posunout o kousek dále, k zásadnímu pojmu třída.

1.2.2. Třída

Už jsme si řekli, že **Igor je objekt**. Že Igor je jedinečný. Ale je tak jedinečný, že bychom ho nemohli k někomu připodobnit? Co o něm víme? Že Igor je také člověk. Jako já, jako vy. Všichni máme mnoho společných vlastností. Všichni jsme nějak vysocí, všichni máme nějakou váhu, hodnotu cholesterolu, každý umíme mluvit, ale mluvíme všichni stejně? Rozhodně ne. Co z toho plyne:

1. Existuje mnoho lidí. Jsou různí, ale mohou mít společné vlastnosti;
2. Každý jsme vytvořený podle společného vzoru, každý máme DNA, ale nikdo stejnou.

Všichni známe DNA, platí? Kdo ne, dohledá si na (něco)pedii. Představme si, že je to šablona, podle které jsou vytvářeni konkrétní lidé. Smícháním určitých genů vznikne určitý člověk. Ještě jednou: je mnoho lidí, ale Igor Novák je jen jeden. Všichni lidé jsou ale vytvořeni podle společného vzoru, podle společné šablony. Tím se dostáváme k jádru věci.

Vzor? Šablona? Pojďme použít lepší termín: **třída**. Třída je papír, na kterém je napsáno, že každý člověk má být nějak vysoký, má mít nějaký věk, má mít nějaké schopnosti. Jakou výšku? Jaký věk? Jaké schopnosti? To třída netuší. Je to opravdu jen šablona. Jeto prázdný formulář. Má kolonky, ale ty kolonky nejsou vyplněné. Až když je vyplníme, vznikne konkrétní člověk. Vznikne náš Igor.

Tohle je zásadní moment. Tvrdíme, že:

1. Igor je objekt (to už jsme tvrdili dříve, že?);
2. Že Igor je člověk, který je vytvořený podle šablony, vzoru, formuláře „člověk“.

Igor je tedy unikátním a neopakovatelným případem člověka. Vyplnili jsme formulář člověk a vznikl Igor. Vyplnili bychom ho jinak a vznikl by Pepa, ještě jinak a měli bychom tu Janu, úplně jinak a měli bychom tu Jamese T. Kirka. Igor je tedy instancí třídy člověk. Co je instance třídy? Je to vyplněný formulář, je to objekt. Objekt je tedy pojem totožný s instanci třídy.

1.2.3. Člen třídy

Tahle pasáž bude stručná. Už víme, kdo je to Igor. Už víme, co je to člověk. Člen třídy nám vypovídá o tom, o jakého člověka se jedná. Je-li člověk formulář, pak člen třídy je konkrétní položkou.

Členové třídy, ale mohou být velice různí:

1. Mohou říkat, že člověk má mít nějakou výšku, věk či barvu očí;
2. Mohou říkat, že člověk může nějak měnit svůj stav, že člověk nějak koná;
3. Mohou říkat, že člověk může nějak reagovat na nějaké události.

Pojďme krok za krokem. První případ nazýváme **vlastností, nebo atributem** (popisujeme, jaký — a pozor zase se jedná až o konkrétního jedince — člověk je). Igor má modré oči. Druhý případ nazýváme **metodou** (popisujeme, jak se daný člověk projevuje). A zase: každý člověk umí hubnout, ale pouze Igor umí hubnout o 5 kg za půl roku. A reakce na události? Představte si, že vás někdo udeří. Každý člověk na to bude reagovat, ale pouze Igor uteče rychlostí 20 km/h směrem na severovýchod za hlasitých výkřiků. Zde jsme se dotkli pojmu **událost** (specifický případ vlastnosti).

1.3. Dědičnost mezi třídami

Dosud jsme se pohybovali na výrazně abstraktní rovině. Pojďme se tedy (mimo jiné) podívat, jak by se výše zmíněné záležitosti zapsaly v programovacím jazyce C#. Nejprve ale stručně k dědičnosti. Řekli jsem si, že Igor (objekt) je člověk (třída). Je jen jeden náš konkrétní Igor, ale je mnoho lidí. Co ale kdybychom je (myšlené lidi) mírně zúžili? A co kdybychom tvrdili, že Igor má přeci jen s nimi něco společného? Pojďme na to:

1. Igor Novák je student. Existuje jen jeden student? Rozhodně ne. Student tedy bude třída. Studentů může být mnoho. Ne tolik jako lidí, ale stále mnoho.
2. Igor Novák je student střední školy. Existuje jen jeden student střední školy? Ne. Student střední školy tedy bude opět třída. Studentů střední školy bude zase o něco méně než studentů obecně, ale pořád se nejedná o žádné unikáty.

Tohle nám ale plodí mnoho, různě sofistikovaných, otázek:

1. Nemá student střední školy mnoho společného s jakýmkoliv studentem? (Vskutku ano.)
2. Nemá student mnoho společného s jakýmkoliv člověkem? (Opět ano.)

Student střední školy (třída) tedy bude mít mnoho společných vlastností (atributů) a schopností (metod) se studentem obecným. A co mnoho, všechny! Obecný student bude pak mít mnoho společných vlastností a schopností se člověkem (a opět nejen mnoho, ale všechny!). Zde se dostáváme k **dědičnosti**. Student střední školy dědí (vlastnosti a schopnosti) od studenta obecného. A student obecný dědí totéž od člověka. Příliš obecné? Pojďme to rozklíčovat:

1. Člověk má věk, výšku a váhu. Každý? Bezpochyby.
2. Má student věk, výšku a váhu? Opět bezpochyby. Student je tedy dědí od člověka. Student dále má navíc studijní průměr a dochází na různé předměty.
3. Má student střední školy studijní průměr a předměty? Ano, opět ano. Každý student střední školy má tyto atributy. Student tedy dědí od obecného studenta. Má ale každý student maturitní zkoušku? Nebo studuje každý student na střední škole? Ne? Pak bychom museli náš model dále upravit.

[uml] | *uml.png*

Shrňme to. Student střední školy dědí (dědí ve smyslu přejímá stejné členy, čili vlastnosti a metody) od studenta obecného. A student obecný dědí od člověka. Od čeho by mohl dědit člověk? Třeba od savce (každý člověk je savec a má s nimi mnoho společného). S čím má něco společného savec? Se zvířetem. S čím zvíře? S jakoukoliv živou formou. Vidíte? Jsme stále obecnější a obecnější. A proto: není jedna živá forma (třída), není jedno zvíře (třída), není jeden člověk (zase třída!). Ale je pouze jeden (pro změnu) Láďa Novák, jakožto unikátní případ studenta.

1.4. Zapouzdření

Další tajemný pojem. Dotýká se ale vskutku prostých věcí. Zůstaňme u našeho Igora. Víme, jak je vysoký? Víme (nebo snadno zjistíme). Víme, jak se chová? Víme. Dovedeme zjistit jeho reakce na konkrétní situace? Vskutku, Pavlov by mohl povídat, že. Zatím jsme se bavili o tom, jak se nám Igor (objekt, pamatujeme?) jeví navenek. Jaký je ale Igor uvnitř? A zajímá nás to vůbec?

Na tomto místě se opět dostáváme k podstatě věci. Zajímat nás to může, ale nemusí:

1. Co když chceme pouze spolupracovat s Igorem (stále objekt) na konkrétním úkolu? Pak nás zajímá, jak dobré má pracovní výkony, jak se projevuje v rámci kolektivu, jak reaguje. Zajímá nás jeho rozhraní (ano, to je už termín z objektového programování stvořený pro tento účel).
2. Zajímá nás ale, jaký je Igor uvnitř? Igor (objekt) je odvozen ze člověka (třída) a jako kterýkoliv jiný, je unikátní. Ale k tomu podstatnému. K čemu vědět jak pracují jeho játra? Srdce? Jaký je jeho krevní tlak?

U objektového programování nás zajímá, jak se kdo (konkrétní objekt) chová a jaké má vlastnosti. Proč se tak chová a proč má jaké vlastnosti, je nám lhostejné. Co když ale sami vytváříme nějaký určitý (ha, ne objekt, ale jeho popis, čili co? Ano, třídu!)? Pak musíme přesně určit, proč se co a jak chová a jaké je to uvnitř. Jaké vlastnosti a schopnostmi má (třeba člověk, byli-li bychom schopni ho vytvořit, že) skryté uvnitř a **kterými se má naopak projevovat navenek**.

To je podstata **zapouzdření**. Každá třída má vlastnosti a metody (kolonky, vzpomínáte), které jsou dostupné buď čistě uvnitř ní samotné (pak bychom je nazvali privátní, popř. označili jako `protected`), nebo takové, kterými se na základě třídy vytvořený objekt projevoval navenek. Ty by pak tvořili jeho rozhraní (a museli by být tzv. veřejné). Zdůrazněme to znovu, třída (třeba člověk) se navenek nijak projevovat nebude, ale Igor, jakožto instance této třídy (objekt), už může být moc zajímavý (mít vlastnosti a metody).

Jasné? Pojďme si to zkomplikovat :-)

1.5. Co jsou to modifikátory přístupu

Co označujeme za členy třídy už víte. Zapouzdření znáte také. Každému členovi třídy je možné nastavit určitý modifikátor přístupu. Zamyslete se nad tím spojením. Modifikujeme přístup. Říkáme, **odkud ke členu dané třídy smí přistoupit** a odkud ne. Běžně se s nimi setkáváte (v C# prakticky ani nemůžete jinak), už od prvního spuštění vývojového prostřední vidíte slova jako `public`, `private` a mnohá jiná. Nyní se podíváme na ty nejpodstatnější ze základních (a zároveň i nejčastěji používaných).

1. K čemu slouží `private`? Použijete ho, když chcete docílit toho, aby daný člen byl dostupný pouze v rámci třídy a dost. Vytvoříte na základě třídy objekt? Pak člen nebude dostupný navenek. Podědíte ze třídy? Opět smůla, nedostanete se k němu.
2. Jak je to u `protected`? Podobné jako v předchozím případě, ale s důležitým rozdílem. Člen bude dostupný ve všech třídách, které ze třídy, kde je umístěn váš člen, dědí. Bude dostupný i navenek? Stále ne, k tomu je třeba další z našeho výčtu.
3. A to konkrétně modifikátor přístupu `public`. Je samopopisný: je veřejný. Takovýto člen bude vždy poděděn a zároveň bude vidět navenek u na základě třídy vytvořeného objetu. Ano, bude tvořit jeho **rozhraní**. Právě v duchu kapitolek o pár řádků výše.

```
/* Demonstrace modifikátorů přístupu */

namespace NeskodnyCsharp
{
    class Clovek
    {
        private int ID; // Vlastnost je přístupná pouze ve třídě Clovek
        protected int Vyska;    // Vlastnost je možné zdědit
        public int Vek; // Vlastnost může tvořit rozhraní objektu
    }

    class Student : Clovek
    {
        public Student()
        {
            Vyska = 188;    // Lze přistupovat k poděděné vlastnosti
            /* Inicializace nového objektu na základě třídy Clovek */
            Clovek muz = new Clovek ();
            muz.Vek = 27;   // Přístup k vlasnoti 'Vek' tvořící rozhraní objektu
        }
    }

}
```

Vždy, když nějaký prvek chcete použít, zamyslete se nad tím, zda ho aktuálně použít můžete. Není-li modifikátor přístupu určen, automaticky se použije `private`, čili nejpřísnější možná varianta. Pozor na to, jedná se o odlišnost od mnoha jiných jazyků (jako třeba Java), kde mohou být jiné zvyklosti.

Pakliže váháte nad tím, který modifikátor přístupu u vámi vytvářeného členu použít, odpovězte si na otázku: Kde a kdo musí nezbytně tento prvek vidět? Snažte se nastavit tuto úroveň na spíše přísnou než naopak. Snáze opravíte chybu, když někomu (nebo něčemu) dáte oprávnění málo (věřte, on se ozve), než chybu, kde oprávnění k přístupu byla až příliš široká (ten údržbář asi opravdu neměl mít přístup k ovládání reaktoru...). Prostě, buďte opatrní (neslyšíte to naposledy).

Chapter 2. Jaké známe druhy tříd

2.1. Popis jednotlivých tříd

S pojmem třída jsme se tedy už střetli. **Vzor, šablona, popis**. To jsou slova, která udržte v paměti. Budeme je potřebovat. Na řádcích o kousek výše jsme si popsali, jak se třída může chovat. Jednalo se o klasický případ, kdy na základě „vyplnění formuláře“ vytváříme konkrétní objekt. Takovou třídu si můžeme nazvat řekněme „klasickou“. Mnoho o ní ale zatím nevíme.

2.1.1. Klasická třída

Obecná, obyčejná, prostě základ, ze kterého vycházíme. Stále rozvíjíme situaci popsanou výše. Pojďme se podívat na to, kde a jak danou třídu definovat, a jak se přiblížit její implementaci v programovacím jazyce C#. Jistě jste si všimli, že když založíme nový projekt, dojde k vytvoření minimálně jednoho souboru, ve kterém se vyskytuje jmenný prostor (`namespace`), uvnitř něhož je — typicky — definována jedna třída. Co s tím?

Jmenný prostor je oblast, ve které na sebe jednotlivé třídy "vidí". Když vytvoříme jmenný prostor `Maturita` a uvnitř něho třídy `Komise`, `Student` a `Zkousejici` mohou na sebe (bez dalšího) bez problémů vidět. Existovat vedle sebe a čekat než někdo vytvoří jejich instance, čili objekty. Na základně třídy `Maturita` např. objekt maturita2015, na základě třídy `Zkousejici` např. `kotekDrsnyUcitel` a podobně.

Rozveďme si odpověď na otázku, kde třídy definovat:

1. Nejčastější situace je, ce které používáme jeden soubor, uvnitř něhož je definována právě jedna třída, která se nachází v rámci jednoho jmenného prostoru. Jsou programovací jazyky, které to ani jiným způsobem neumí a vyžadují právě striktní situaci aneb jedna třída uvnitř jednoho prostoru (a má to svou logiku a smysl např. z hlediska přehlednosti).
2. Druhá možnost je mít uvnitř jednoho souboru v *jednom jmenném prostoru více tříd*. Typicky je to vhodné v situaci, kdy chceme tyto jednotlivé třídy využít relativně jednoduchým způsobem. Naopak si představte, že byste měli dvě třídy v jediném souboru a chtěli je souběžně editovat. Vskutku, to není hezká představa skákat neustále, řekněmě, z řádku 353 na řádek 27, není-liž pravda?
3. A pak je tu situace třetí. Můžeme mít dva soubory sdílející jeden a ten samý jmenný prostor, ve kterých je obsažena právě jedna (ano jedna jediná) třída. Bavíme se pak, poměrně logicky, o tzv. částečné — partial — třídě. I tato situace může mít spoje opodstatnění. Poměrně běžný je případ, kdy v jedné partial třídě je generovaný kód (pokud jste někdy pracovali, ve Windows Forms, znáte to dobře) a ve druhé kód, kterým pak ručně rozšiřujeme (doplňujeme) kód ze souboru prvního. Důležité je si uvědomit, že pouze dohromady je vytvořena jedna celistvá třída s jedním názvem.

```
/* Demonstrace částečných 'partial' tříd */

namespace NeskodnyCsharp
{
    /**
     * Každá z 'partial' tříd může být v jiném souboru,
     * ale musí být ve stejném namespace
    */

    partial class Clovek
    {
        public int Vek;
    }

    partial class Clovek
    {
        public int Vyska;
    }

    class Program
    {
        public static void Main()
        {
            /* Inicializace nového objektu na základě třídy Clovek */
            Clovek muz = new Clovek();
            /* Lze přistupovat k oběma vlastnostem z obou 'partial' tříd */
            muz.Vek = 27;
            muz.Vyska = 189;
        }
    }
}
```

Co si z odnést? Předně to, že není jedno správné řešení. Není žádné dobré versus špatné. Lepší nebo horší. Zahoďte tento způsob uvažování a omezte se na mnohem elegantnější: vhodné a nevhodné. Všechny zmíněné třídy mají své opodstatnění a své využití. Dbejte na to.

2.1.2. Abstraktní třída

Je třída, ze které nemůžete (a nechcete) vytvořit její instanci, **nemůžete vytvořit objekt**. Abstraktní třída je výrazně neužitečná bez toho, aby z ní někdo (nějaká jiná třída) dědila. Představte si, že máme třídu `Firma`. V každé firmě budeme chtít počítat platy, třída `Firma` by tedy měla mít logicky metodu (ano metodu, podívejme se o pár odstavců výše), která slouží k takovému výpočtu. Něco na způsob `VypoctiMzdy` (jak by se mohla jmenovat).

Položte si teď zásadní otázku: Víte, jak se budou počítat mzdy u každé jednotlivé firmy? Stejně? Sotva. Systém prémií, benefitů apod. bude v různých oblastech dost jiný. Takže pozor. Tvrdíme, že každá firma potřebuje nějakým způsobem mzdy počítat, ale že těžko nalezneme nějaký jednotící

model. A to je ono! O tom jsou abstraktní třídy. Nemůžeme vytvořit objekt na základě třídy `Firma`, ale můžeme pevně předepsat, že každá třída, která z ní bude dědit, mzdy (už konkrétním zpsůobem) počítat musí. Musí obsahovat metodu `VypoctiMzdy`. Předepišme to tedy. Kde? No právě v abstraktní třídě. Která to je? V tomto případě právě třída `Firma`. Takže platí, že:

1. Abstraktní třída `Firma` bude obsahovat deklaraci (opět abstraktní) metody `VypoctiMzdy`. Pouze deklaraci (čistý fakt, že má existovat), nikoliv definici (jak to má konkrétně počítat). Ještě jednou: na základě této třídy nemůžeme vytvořit instanci (objekt).
2. Třídy (teď již „obyčejné“), které se mohou jmenovat např. `Automobilka` a `Reklamni`, které budou dědit z třídy `Firma`, pak budou muset každá obsahovat jinou definici metody `VypoctiMzdy`. Muset jako muset. Nebudou-li ji obsahovat, projekt ani nepůjde zkompilovat. Hotovo. Jak prosté, milý Watsone.

Abstraktní třídy tedy nebyly právě intelektuální výzvou, pojďme na třídy statické :-)

2.1.3. Statická třída

Neslyšíte to už v jejím názvu? Statická třída. Neměnná třída. Hotová třída. Ani z této třídy **nevytváříme objekty**, u této třídy to hlavně ani dělat nechceme. Měli jste někdy potřebu vypsat nějaká data do konzole v přesně definovaném formátu? Nebo vícenásobně použít nějakou metodu, kterou jste jinde definovali? Tady jsme u toho. Opět si položte otázku: Má smysl, abyste definovali identickou např. metodu `VypisVeSloupcich` v deseti různých třídách? Asi řeknete ne, ale vyvstává tu logická námitka: Proč nemohu tuto metodu definovat ve třídě, ze které ostatní dědí?

Toto je zásadní moment. Proč ne? Vždyť to má svou logiku. Uvědomme si následující. Pokud mezi sebou třídy dědí, znamená to, že mezi sebou mají vztah, logickou návaznost, propojenost vztahu od obecného ke konkrétnímu. Od třídy `Firma` může dědit třída `Automobilka`, od třídy `Automobilka` třída `CeskaAutomobilka`. V pořádku? Dobrá. Asi se shodneme, že každá z těchto tříd by mohla chtít umět vypsat nějaká data ve sloupcích. Stále v pořádku? OK. Znamená to ale nutně, že třída `Pracka` dědící ze třídy `Vyrobek`, žádná taková data k vypsání nemá? Aha, jsme u toho.

Problém je následující:

1. Metoda `VypisVeSloupcich`, která by byla obsažená ve třídách `Firma` a `Vyrobek`, by dělala přesně to samé a přesně ve stejných situacích. Pracovala by přesně se stejnými vstupními daty. Byla by stejná. Opakování se, opakování se, opakování se. To je přesně to, co nechceme. Pamatujme na to. Vždy, když narazíme na opakování, má nám to značit, že je něco špatně (a bohužel dost typicky a pravděpodobně na naší straně, to jest hoře programátorovo).
2. Správný postup je, jak název kapitoly napovídá, použít **statickou** třídu. Statická třída obsahuje výčet obecně použitelných metod nebo konstant, které chceme použít nezávisle na dalším. Naše statická třída by se tedy mohla jmenovat např. `Vystup` a metoda `VypisVeSloupcich` by jako argument mohla přebírat data jakéhokoliv objektu, která by bylo třeba v nějaký okamžik vypsat.

Statická třída je tedy o výčtu. O znovupoužitelnosti. O použitelnosti nezávisle na určitých datech. Vypadá tajemně, ale není. Znáte třídu `Math`(měli byste)? Obsahuje metody pro výpočet sinu a konstanty, jako je např. PI. Zkoušeli jste si někdy vytvořit instanci třídy `Math`..? Asi tušíte kam směřuji, nešlo by to, je statická. Vztahuje se výpočet sinu a PI k jedné konkrétní oblasti? Ne? Oba prvky ale jistě použijeme v mnoha jiných situacích (konstrukce, analytická geometrie, energetika), až v desítkách či stovkách. Ne v jedné oblasti. Řekli jsme mnoho a vychází nám klasický kandidát na statickou třídu.

2.2. Klíčové pojmy k třídám

2.2.1. Konstruktor

Zatím jsme se drželi jen u těch nejzákladnějších termínů, pojďme se podívat na některé lehce specifičtější. Co je to konstruktor? Existuje na to extrémně stručná odpověď: metoda. Konstruktor je nicméně **metoda, která je velmi zvláštní**, a to následujícími věcmi:

1. Jmenuje se stejně jako třída, ve které je obsažena. Třída `Skola`, tedy bude obsahovat konstruktor s názvem `Skola()`.
2. Nemá žádný návratový typ. U obyčejné metody musíme určit minimálně už to, že metoda žádnou hodnotu nevrací (použijeme pak klíčové slovo `void`). U konstruktoru ne.
3. Je volána v okamžiku, kdy vytváříme konkrétní instanci třídy. Ano píšeme-li `Skola ssps = new Skola()`, pak výraz `Skola()` je konstruktorem této třídy.

K čemu je nám v praxi konstruktor dobrý? Znovu se vraťme k tomu, k čemu je nám třída. Je to prázdný formulář, který chceme vyplnit. Klíčové je ale *kdy*. Pokud bychom vytvořili objekt podle příkladu výše (v bodě 3), pak je to sice formulář s razítkem, ale stále poněkud smutně prázdný. Objekt existuje, ale je takový lejaký nijaký. Co s tím?

1. Můžeme postupně přistoupit k veřejně přístupným (tedy `public`) vlastnostem objektu (tvořícím rozhraní objektu, pamatujeme?) a postupně určit, že `ssps` má takovéhoto ředitele, takovéto studenty apod. To je cesta první.
2. Nebo můžeme tyto údaje předat přímo konstruktoru jako argumenty. V okamžiku, kdy někdo vytváří objekt na základě třídy škola, je tímto donucen, aby alespoň minimální množství argumentů předal a prázdno vyplnil.

```
/* Jak lze použít konstruktor */

namespace NeskodnyCsharp
{
    class Skola
    {
        /* Třídní proměnné Nazev a Adresa */
        public string Nazev;
        public string Adresa;

        /* Konstruktor třídy a jeho typické použití */
        public Skola(string naz, string adr)
        {
            /**
             * Přiřazení výchozích hodnot do třídních proměnných
             * pomocí přiřazení v konstruktoru
            */
            Nazev = naz;
            Adresa = adr;
        }
    }

    class Program
    {
        public static void Main()
        {
            /**
             * Vytvoření nového objektu 's' dle konstruktoru Skola()
             * třídy Skola s jeho argumenty
            */
            Skola s = new Skola("SPS", "Adresa 25, Praha");
        }
    }
}
```

Protože programování není o uctívání svatých krav, můžeme oba přístupy kombinovat, třeba použít konstruktory více s různými požadovanými argumenty, pak se bavíme o přetěžování konstruktoru. (Chcete vědět víc? Podívejte se na kapitolu „Proč přetěžovat metody“.) Můžeme tak klidně definovat konstruktor, který nepřebírá žádný argument; který přebírá argument jeden, nebo další, který jich přebírá více. Co když ve třídě žádný konstruktor nepoužijeme? Je to prosté, pak se použije implicitní konstruktor, který sice nevidíme, ale je přítomen.

2.2.2. Slůvka „this“ a „base“

Co znamenají? Je dost pravděpodobné, že jste se s nimi někde v rámci tvořeného zdrojového kódu setkali. Neskrývá se za nimi nic tajemného. Běžně v rámci nějaké třídy voláme (nebo přistupujeme k) její člen. Nejčastěji ho voláme čistě jeho názvem (a v C# není mnoho situací, které by nás nutili k tomu dělat to jinak). Vrátíme-li se k naší třídě `Clovek` vidíme, že jsme pracovali se členem (zde atributem) značícím výšku (nazvěme ho tedy `Vyska`). Můžeme ho v rámci třídy volat jednoduše, tedy

pouze `Vyska`, jako by to byla jakákoliv jiná proměnná. Stejně tak dobře ale můžeme napsat `this.Vyska`. Proč je tomu tak?

1. Slovo `this` nám ukazuje na konkrétní instanci třídy, tedy objekt. Můžeme mít mnoho různých instancí třídy `Clovek` (Igora, Honzu, kohokoliv) a každá z nich v sobě bude obsahovat vlastnost `Vyska`. Zápisem `this.Vyska` říkáme, že se vztahujeme právě k tomuto konkrétnímu člověku a atributem `Vyska` k hodnotě poplatné právě a jen tomuto člověku.
2. Můžeme si to demonstrovat ještě jinak. Program, který není spuštěný, leží na pevném disku. Platí? V okamžiku, kdy ho spustíme, se stane procesem a získá svá specifika (přidělenou operační paměť, prioritu procesu) včetně konkrétních hodnot jeho atributů. V čem je klíčový fígl? Daný program můžete (typicky) spustit vícekrát, jedná se tedy o více spuštěných instancí (a slůvko `this` se vztahuje ke každému jednotlivému z nich).

Jak byl popis slůvka `this` vyčerpávající, popis slůvka `base` bude o to kratší. Týká se dědičnosti. V C# máme vždycky maximálně jednu třídu, ze které dědíme, ne více. A pojem `base` se vztahuje právě k ní. Může nastat situace, kdy budeme potřebovat sáhnout na nějaký člen obsažený právě v této třídě a přesně k tomu se `base` hodí. Ostatně to je přesně ten důvod, proč dané třídě říkáme *bázová*.

Pojďme si možné situace ilustrovat na příkladu:

1. Třída `Student` dědí ze třídy `Clovek`. Uvěřitelný předpoklad. Budu-li si chtít sáhnout v rámci kódu ve třídě `Student` na atribut `Vyska` ve tvaru, ve kterém je obsažený ve třídě `Clovek`, napíšu poměrně přímočaře `base.Vyska`. Lehké, že?
2. Vaše úplně logická otázka pravděpodobně je, proč bychom něco takového vůbec měli dělat? Vždyť jsme obsah třídy `Clovek` podědili (ano, máme-li nastavené modifikátory přístupu minimálně na `protected`). Máte pravdu. Vtip je v tom, že ne vždy chceme v aktuální třídě použít podděděný člen přesně v takovém tvaru, ale chceme pracovat s oběma.
3. Zmatečné? Možná to tak na první pohled vypadá, ale všechno bude křišťálově čisté hned, jak se dostane k další kapitole o polymorfismu. Pro teď si zapamatujme, že s `base` se asi nejčastěji v C# setkáme, když chceme přebrat ze třídy, ze které dědíme, konstruktor a třeba ho mírně doplnit a rozšířit. Tajemné? Není, podívejte se na příklad níže.

```
/* Jak prakticky využít 'base' a 'this' */

namespace NeskodnyCsharp
{
    class Clovek
    {
        public int Vyska;

        public Clovek(int vys)
        {
            /* Oba dva zápisy dělají naprosto totéž */
            this.Vyska = vys;
            Vyska = vys;    // Zkrácený tvar přiřazení
        }
    }

    class Student : Clovek
    {
        public string Skola;

        /**
         * Podědění bázového konstruktoru a jeho rozšíření
         * o další argument - určení školy
        */
        public Student(int vys, string sko) : base(vys)
        {
            Skola = sko;    // Opět, lze zapsat i jako this.Skola = sko;
        }
    }
}
```

Vidíte, že na tom nic není. Námi nově vytvořený konstruktor (ve třídě `Student`) zbytečně **neopakuje, co už dělá konstruktor v bázové třídě** `Clovek` (to slůvko si pamatujme). Přebírá argumenty poplatné bázovému konstruktoru a předává mu je, zatímco sám nově vyžaduje pouze argument `skola`, který předává do třídní proměnné `Skola`. Pouze a jen to, žádný zbytečný kód navíc. (A co si budeme říkat, pouze a jedině tam má smysl. Chodí snad každý člověk stále do školy?)

Chapter 3. K čemu slouží a co je to polymorfismus

3.1. Proč přetěžovat metody

Protože chceme pro metody používat hezké a dobře **zapamatovatelné názvy**. A spolu s tím — protože naopak nechceme používat mnoho podobných (a lehce zaměnitelných) názvů. To je podstatou. Z čeho ale vyplývá taková potřeba? Můžeme chtít mít v našem programu metody, které dělají principiálně totéž, ale přebírají různý počet argumentů. Jak by to tedy mohlo vypadat?

Pojďme přijít na nějaký příklad:

1. Metoda, která vypisuje počet sekund na základě vstupních dat, by se mohla nazývat, řekněme, `VypisSekundy`. Co by taková metoda mohla přebírat na vstupu jako argument? Dny? Hodiny? Minuty? Dny a hodiny? Dny, hodiny a minuty?
2. Pro každou z těchto situací bychom dovedli vymyslet hezký název. Něco a la `VypisSekundyNaZakladeHodin`, `VypisSekundyNaZakladeHodinMinut`. Už teď si je nepamatujete, že? Přitom původní název `VypisSekundy` je zapamatovatelný dokonale. A tady nastupuje přetěžování, my tento název vícenásobně použít opravdu můžeme.
3. Pozor, **přetěžování není všespásné**. Použijeme ho, když máme různý počet argumentů, nebo když máme argumenty jiného typu. Kdybychom chtěli přetížit název metody `VypisSekundy` a na vstupu předávat jako jeden argument v prvním případě počet minut (jako celé číslo), ve druhém pak počet hodin (zase jako celé číslo), máme smůlu.

Chtěl jsem napsat dlouhý vysvětlující odstavec, ale podívejte se raději o pár odstavců níže. Není tam uvedený kód jasný? Máme hezké zapamatovatelné názvy metod? Máme. Mají metody různý počet argumentů? Opět ano. Takže jsme právě viděli podstatu přetěžování. A o tom to je.

3.2. Kdy použít překrývání

Všimněte si, že dosud jsme se pohybovali pouze v rámci jedné třídy. Všechny přetížené metody tam byly obsaženy hezky pěkně pod sebou. Tak to ale být nemusí. Je úplně běžná situace, že chceme nějakou metodu podědit, ale použít jinak. (A ano tohle je přesně to místo, které by vás mělo začít zajímat po odskoku od nyní již známého slůvka `base`.) Může to být libovolná metoda, klidně včetně konstruktoru.

Klasikou v tomhle směru je v C# metoda na převod čehokoliv na řetězec:

1. Aneb `ToString`. Naleznete ji u jakéhokoliv objektu, takže v podstatě u všeho. Proč? Její definice je obsažena ve třídě `Object` a z této třídy dědí doslova všechny další třídy. Všechny. Na počátku je vždy ona. Chová se ale metoda `ToString` stejně u objektu typu `Int32` i u, řekněme, objektu typu `Button`?
2. Odpověď asi tušíte. Ne, chovají se jinak. Jak převést číslo na řetězec? Jednoduše. Ale tlačítko na řetězec? Uff, to už vypadá na netriviální problém. Logicky postupně dospějeme k tomu, že jak ve třídě `Int32`, tak `Button`, bude mít metoda `ToString` mít svou definici, ale ta bude zároveň v každé třídě jiná. Jiná, i když obě dědí ze třídy `Object`. A zde nastupuje překrývání.

Kdy tedy překrývání použít? Vždy, když chceme použít nějaký název znovu v rámci hierarchie dědičnosti. A z dobrého důvodu. Používejme tuto možnost vždy, když víme, co děláme, jinak se stane zbraní hromadného ničení. (Ale lze celkem úspěšně předpokládat, že taková raketa typu Trident bude programována spíše v Adě než v C#.) Pohybujeme se tedy po linii dědičnosti, vertikálně. Horizontální pohyb by zde byl čím? Správně, přetěžováním.

3.2.1. Virtuální a abstraktní metody

Máme tedy bázovou třídu, ze které dědíme. V té máme metodu, jejíž název chceme dále — ve třídě, která z ní bude dědit — použít. Tyto metody chceme nějak označit, dát najevo, že mohou (nebo musí) být ve třídě z bázové dědící předefinovány (nebo definovány). Použijeme pro to dvě klíčová slůvka a to `virtual` a `abstract`. Bavíme se pak o virtuální a abstraktní metodě. Povědomé? Ano. Přesně, už jsme se o tom bavili v kapitole o abstraktních třídách a toto povídání na ni přesně navazuje.

1. **Virtuální metoda** je metoda, u které dáváme najevo, že může, ale nemusí, být překryta. Dáváme, sami sobě, nebo jinému programátorovi, najevo, že se scénářem překrytí takovéto metody se prostě počítá.
2. **Abstraktní metodu** bychom už měli znát. Ta nejenom, že může. Ne, ta musí být definována. V bázové třídě je obsažen pouze popis, že má existovat metoda s určitým názvem a přebírat nějaké argumenty. Je deklarovaná. Její definice ale musí přijít až ve třídě, která z ní podědí.

Asi vás napadnou dva možné scénáře:

1. Co když slůvko `virtual` neuvedu? Moderní kompilátory do bytecode (nebo přesněji do CIL) vám to pravděpodobně povolí a funkčnost bude opticky identická, nicméně nedělejme to. Je to příklad benevolence, který se nemusí vždy vyplatit.
2. Co když metodu, která je uvedena jako abstraktní, dále (po podědění) nedefinuji? Tady to je prosté. Váš kód nepůjde zkompilovat. Hotovo. Chybové hlášení, které to popisuje, je ale velmi přesné. Nemusíte se bát. Uvidíte a opravíte.

3.2.2. Logika klíčových slov override a new

Zatím jsme se pohybovali na úrovni definic (nebo deklarací) metod, které můžeme (nebo musíme) překrýt. Jak je to ale na druhé straně? U tříd, které z nich dědí? Stejně tak, jako dáváme najevo, že povolujeme překrytí metody, musíme dát najevo, že danou metodu překrýváme (nebo definujeme). I tady máme klíčová slova, nyní v podobě pojmů `new` a `override`. Jak na ně?

1. Použijeme-li klíčové slovo `override`, říkáme, že metoda, kterou právě chceme napsat, bude zachovávat logiku té, kterou překrýváme. Neměníme **její smysl**. Neměníme její **návratový typ**. `ToString` by vždy měla převádět něco na řetězec. Ne vždy to ale bude dělat tím jediným správným způsobem. Proto překrýváme. To dává smysl, že?
2. A co když je to „nové“? Kdy použít `new`? To je vzácnější situace. Někdy můžeme chtít použít už existující název metody pro zcela novou věc. Ve zcela nové situaci. Pro metodu, která se **liší od oné překrývané smyslem i návratovým typem**. Představte si, že by metoda `ToString` vracela celé číslo, které by odpovídalo nějakému matematickému výpočtu. To by ale nedávalo smysl, že? Dobře si rozmyslete, jestli něco takového chcete udělat.

```
/* Demonstrace překrývání a přetěžování */

using System;

namespace NeskodnyCsharp
{
    class Clovek
    {
        public int Vyska;

        /* Virtuální metoda, kterou je možné přerýt */
        public virtual string VypisUdaje()
        {
            return Vyska.ToString ();
        }
    }

    class Student : Clovek
    {
        public string Skola;

        /* Metoda, která překrývá bázovou metodu 'VypisUdaje' */
        public override string VypisUdaje ()
        {
            return String.Format ("{0} {1}", Vyska, Skola);
        }

        /* Metoda, která přetěžuje metodu 'VypisUdaje' - má jiný počet argumentů */
        public override string VypisUdaje (bool vsechno)
        {
            if (vsechno)
            {
                return String.Format ("{0}, {1}", Vyska, Skola);
            }
            else
            {
                return Skola;
            }
        }
    }
}
```

A opět si rozebereme příklad. Pozorně se podívejte na komentáře, které jsou v něm obsažené. Překrývání nám obecně může ušetřit mnoho práce, může usnadnit a zpřehlednit náš kód už tím, že v něm budou použity komukoliv známé názvy metod, které budou dělat správně to, co se od nich běžně očekává. Ale ještě jednou zopakuji. Opatrně. Používejte překrývání, jen když má smysl.

Chapter 4. Jak se pracuje s rozhraními?

4.1. Význam rozhraní

Aneb mikrokapitola na závěr. S pojmem rozhraní jsme se už setkali. Můžeme ho použít v obecném smyslu, právě tak jsme o něm ostatně již mluvili. Objekt má nějaké rozhraní. Klasika. V programovacím jazyce C# však ale máme ještě další nástroj pro to, jak s rozhraními pracovat. Jakýsi předpis, který nám říká, že musíme v dané třídě implementovat konkrétní metody. Tento předpis se nazývá **interface**, čili do češtiny přeloženo: rozhraní. Jestliže vám to připomíná pojetí abstraktních tříd, je to zcela správná úvaha, nicméně rozhraní nejsou přesnou analogií. Proč tomu tak je?

1. Vzpomeňte si, že v C# můžete **dědit pouze a výhradně z jedné třídy**. Tečka. Včetně abstraktních tříd. To je velmi omezující. Chtěli byste podědit z naší známé třídy `Clovek` a zároveň z abstraktní třídy `Chovani`? Smůla, jakkoliv by to mělo svou logiku.
2. Rozhraní nicméně můžeme **implementovat tolik, kolik chceme**. Kdyby naše třída měla dědit ze třídy `Clovek` a implementovat rozhraní `Chovani`, šlo by to? Bez potíží. A klidně by mohla pokračovat implementováním dalších rozhraní. A že u člověka si lze představit mnoho předpisů, které každý jednotlivec musí splnit…

```
/* Demonstrace použití rozhraní */

using System;

namespace NeskodnyCsharp
{
    interface Chovani
    {
        /* Sada metod, které musí být implementovány */
        void Mluvit ();
        void Myslet ();
    }

    class Clovek
    {
        public string Jmeno;
    }

    /**
     * Třída 'Student' dědí ze třídy člověk a zároveň implementuje
     * předepsané rozhraní 'Chovani'
    */
    class Student : Clovek, Chovani
    {
        /* Nezbytné definice předepsaných metod */
        public void Mluvit()
        {
            Console.WriteLine (Jmeno + ": Mluvím!");
        }

        public void Myslet()
        {
            Console.WriteLine (Jmeno + ": Myslím!");
        }
    }
}
```

Příklad výše nám tedy jasně ukazuje, o čem rozhraní jsou. Bylo nařízeno implementovat určitou sadu metod? Bylo. Podařilo se tyto metody implementovat? Podařilo. Pak gratuluji, dostali jste se k závěru našeho povídání.

Chapter 5. Závěr

(Ne)škodný úvod do pojetí objektového programování v prostředí programovacího jazyka C# máte nyní za sebou. Ale to by neměl být konec. Zpochybňujte. Dohledávejte další zdroje. Hledejte změny. Porovnávejte se situací v jiných jazycích. Ne všechny programovací jazyky mají svůj objektový model založený na třídách (podívejte se na Javascript a jeho pojetí OOP přes prototypy), ne všechny jazyky mají rozhraní a zakazují vícenásobnou dědičnost (Python). Ne všechny jazyky znají modifikátory přístupu (opět je jich mnoho). A šlo by pokračovat.

V době, kdy tento text čtete, už může být vývoj objektového programování (a zdaleka nejen toho) na místě, který byste vy a já nečekali. A proto Shrnu v jedné větě: **Nebuďte statičtí**, nezůstávejte na místě.