# 

# Objektové programování

Objektové programování je součást tzv. programovacích paradigmat (=vzorec nebo model myšlení), což je několik základních programovacích stylů (paradigma je např. i procedurální programování – třeba C++ umí jak procedurální, tak objektové). Jeho způsob fungování se silně odráží od teorie objektů. Stěžejní částí OOP je snaha kopírovat reálný svět pokud možno 1:1 - použiji příklad z Wiki, dveře otevíráme stále stejně; nezáleží na tom, jestli na dveřích je kukátko, řetízek, jsou bílé nebo hnědé… Z toho vyplývá, že můžeme věci během psaní kódu recyklovat, použít je klidně stokrát znovu, případně je trochu upravit, aby odpovídaly našim potřebám a potřebám programu.

## Teorie objektů

Existuje obecná teorie, která se zabývá objekty, jejich vlastnostmi a vším k tomu přidruženým. Základním pojmem této teorie je **třída**, což je pojem ze všech asi nejobecnější – používáme ho ke klasifikaci procesu, jak uspořádat informace, ze kterých chceme náš výsledek složit, do smysluplných struktur. Podpojmem třídy je **objekt**, který využije předpis stvořený pomocí třídy a stane se takzvanou *instancí třídy*. Objekt má dva základní “smysly svojí existence” - pamatovat si svůj stav pomocí atributů a poskytovat své metody k tomu, aby s objektem šlo zvnějšku pracovat. Když použijeme objekt, zajímá nás, co se stane a jaký bude výsledek, ale v momentě použití už nás nezajímá, jak přesně se to stalo (za předpokladu, že nejste člověk pověřený testováním objektu).

### Dědičnost

Dědičností předcházíme situaci, že pro podobné objekty budeme mít nespočetně tříd. Asi nejznámějším případem je Člověk (berme jako třídu), mohu vytvořit třídu pro sebe a mého učitele, ale není jednodušší vytvořit člověka, který bude mít všechny společné vlastnosti mě i učitele, tj. věk, výšku, jméno, příjmení, a poté mít potomka Studenta se třídou, průměrem a učitele s pracovní pozicí? Ušetříme tím psaní dokola všech stejných atributů. Při dědění dochází k sdílení nejen atributů, ale i metod, s tím, že metody lze v třídě potomka přepsat pomocí *override.* Předpokládáme, že budeme tvořit instance pouze od tříd Studenta a Učitele. Nepotřebujeme-li tvořit instance od obecného Člověka, můžeme třídu označit za *abstraktní*, tím se zajistí, že od ní instance vytvořit nelze.

### Metody

Metody voláme pomocí takzvané tečkové konvence, tak, jak ji důvěrně známe, *instance.metoda().* Vztáhneme-li metody zpět k dědění, je několik způsobů, jak s metodou při dědění naložit. Můžeme ji pouze zdědit a dál jen používat, nebo můžeme použít příkaz *override* (pozn. české zdroje uvádí název překrytí metody, já používám anglický název, takže nevím, jak moc se to vážně používá). Výhodou je, že v překryté metodě můžeme stále volat i původní metodu, protože k ní pomocí dědění máme přístup. Dalším způsobem je *method overloading*, tedy přetěžování metod, kde se název metody shoduje, ale vstupní parametry jsou jiné. Rozdíly v metodách pak můžeme vyřešit až uvnitř metod, ale budeme při volání vybírat různé varianty stejné metody podle potřeby.

### Rozhraní (Interface)

Rozhraní využíváme, pokud zjistíme, že některé naše třídy budou mít společnou metodu (např. by dvě třídy pro člověka uměly to samé, ale měly jiné vlastnosti). Do rozhraní pouze napíšeme hlavičky metod tak, jak to odpovídá syntaxi daného jazyka, a po implementování rozhraní do třídy sepíšeme konkrétní implementaci dané metody.

### Polymorfismus

Je vícero druhů polymorfismu a články o něm jsou dost matoucí, ale tak, jak jsme se s ním setkali my, je to v zásadě možnost ukládat rozdílné objekty navzájem do sebe na základě hierarchie dědičnosti. Jinak řečeno, pokud vytvořím třídu čtyřúhelník a od něj oddědím například čtverec a obdélník, obě dvě děděné třídy mohu uložit jako instance do nějakého čtyřúhelníku; samozřejmě to nefunguje obráceně. Jde to takto například ale i s rozhraními.

### Zapouzdření

Zapouzdření znamená, že k proměnným v nějakém objektu se mohou dostat jen blíže specifikované další části programu. Cílem je, aby při vytvoření objektu jediné, co objekt ukazuje navenek, byly metody a nějaký způsob komunikace se zbytkem (getters, setters, ToString apod). Úrovně přístupových práv se mohou mezi programovacími jazyky lišit, ale nejčastěji se používá (v C#) např. *private = nejvíce omezující, přístup k obsahu třídy/structu je možný pouze uvnitř té stejné třídy/structu; protected = přístup k obsahu pouze uvnitř třídy a v dalších odděděných třídách atd.; public asi není třeba vysvětlovat; existují dále např. internal, protected internal, private protected…*

**public**

The type or member can be accessed by any other code in the same assembly or another assembly that references it.

**private**

The type or member can only be accessed by code in the same class or struct.

**protected**

The type or member can only be accessed by code in the same class or struct, or in a derived class.

**private protected** (added in C# 7.2)

The type or member can only be accessed by code in the same class or struct, or in a derived class from the same assembly, but not from another assembly.

**internal**

The type or member can be accessed by any code in the same assembly, but not from another assembly.

**protected internal**

The type or member can be accessed by any code in the same assembly, or by any derived class in another assembly.

Assembly ... the smallest unit of deployment for any .NET project.

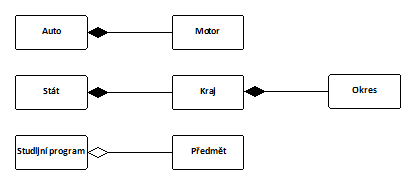
**static modifier**

The static modifier on a class means that the class cannot be instantiated, and that all of its members are static. A static member has one version regardless of how many instances of its enclosing type are created.

A static class is basically the same as a non-static class, but there is one difference: a static class cannot be externally instantiated. In other words, you cannot use the new keyword to create a variable of the class type. Because there is no instance variable, you access the members of a static class by using the class name itself.

## Agregace a kompozice objektů

Agregace a kompozice jsou vztahy mezi třídami. Trochu vtipné je, že oficiálně je oboje agregace, ale agregace = sdílená (shared) agregace, kompozice = složená (***composite****)* agregace. Zůstaneme ale u agregace a kompozice, protože se to tak používá nejčastěji. Oba vztahy jsou mezi nějakým celkem a jeho částí a obě se značí kosočtvercem na straně celku – rozdíl je v tom, že agregace se značí prázdným a kompozice

plným kosočtvercem. 

Zde je vidět, že třídu můžeme zároveň definovat jako celek i část – nemůžeme je ale do sebe zacyklit, nedávalo by to smysl. Platí také podmínka, že v případě kompozice může být část součástí maximálně jednoho celku. U agregace to neplatí, zde může část bez problémů být obsažena ve více celcích (např. předmět a různé studijní programy, učitel učí na více školách apod.)

Agregace

představuje volnou vazbu mezi celkem a součástí, kdy jeden objekt (celek) využívá služby dalších objektů (součástí). Například vztah mezi počítačem a tiskárnou je vztah typu agregace, kdy počítač s tiskárnou tvoří jeden celek, ale tiskárna může existovat i tehdy, pokud není k žádnému počítači připojena. Agregace je formou asociace a v grafické podobě se odlišuje prázdným kosočtvercem na straně celku.

Silnější formou agregace je kompozice. Jde opět o vztah mezi celkem a součástí, ale tento vztah je velmi těsný a neumožňuje samostatnou existenci součásti, aniž by byla připojena k nějakému celku. Navíc na rozdíl od agregace tato součást musí patřit jen jedinému celku a není možné ji sdílet více celky. Příkladem kompozice je vztah mezi fakturou a jejími položkami (viz Obr. 1.7). Každá položka musí být součástí právě jedné faktury, faktura má jednu a více položek. Jestliže fakturu zahodíme, nezbudou nám po ni ani žádné položky.