12. Objektově orientované programování

Created	@October 14, 2024 8:25 AM
Tags	Done
Kdo vypracoval	Mára

Jedla říkal:

abstrakce

zapouzdření

dědičnost

polymorphismus

třídy

instance

Objektově orientované programování (OOP)

Objektově orientované programování (OOP) je **programátorské paradigma**, ve kterém jsou programy koncipovány jako soubor spolupracujících **objektů**. Objekt v OOP představuje entitu (např. reálný předmět nebo koncept), která **obsahuje data i funkce** pro práci s těmito daty. Data objektu nazýváme **atributy** (stav) a funkce definované uvnitř objektu nazýváme **metody** (chování). Objekty se vytvářejí podle definic zvaných **třídy** – každý objekt je **instancí** určité třídy. OOP stojí na několika klíčových principech, mezi které patří **abstrakce**, **zapouzdření**, **dědičnost** a **polymorfismus** (Objektově orientované programování – Wikipedie). Níže jsou jednotlivé pojmy vysvětleny a ilustrovány jednoduchými příklady v Pythonu.

Abstrakce

Definice: Abstrakce znamená **zjednodušení složité reality** odfiltrováním nepodstatných detailů a vyzdvižením podstatných vlastností. V kontextu OOP

jde o to, že se při návrhu tříd soustředíme na **to, co je pro objekty důležité**, a ignorujeme v daném kontextu nedůležité detaily implementace. Každý objekt tak můžeme chápat jako *černou skříňku*, která provádí určité činnosti a komunikuje s okolím, aniž bychom museli znát detaily jeho vnitřního fungování (Objektově orientované programování – Wikipedie). Abstrakce se často projevuje vytvořením obecných tříd či rozhraní, které definují **co** objekty dovedou (jaké metody nabízejí), ale už neřeší **jak** to konkrétně dělají – to ponechávají na specializovanějších třídách.

Účel: Cílem abstrakce je **zjednodušit práci s komplexními systémy**. Programátor může pracovat s objektem skrze jeho rozhraní a **nemusí se starat o vnitřní detaily**. To umožňuje navrhovat modulární systémy – můžeme například definovat obecnou třídu či rozhraní pro *Zvíře* s metodou `vydej_zvuk()`, aniž by nás zajímalo, **jaký zvuk** konkrétní zvíře vydává. Detaily (štěkání, mňoukání atd.) si řeší každá podtřída sama.

Výhody abstrakce:

- Umožňuje **řídit složitost** – můžeme pracovat s koncepty na vyšší úrovni a ignorovat podrobnosti, které pro daný kontext nejsou důležité. Tím se zvyšuje přehlednost a srozumitelnost programu.
- Usnadňuje **opakované použití kódu** – obecné abstrakce (např. rozhraní) lze využít v různých projektech a situacích (Objektově orientované programování – Wikipedie). Nové objekty mohou implementovat existující abstrakce a zapadnout tak do systému.

Nevýhody abstrakce:

- Nevhodně zvolená nebo příliš obecná abstrakce může ztížit implementaci – pokud **abstrahujeme špatné detaily**, může být výsledný model nepraktický a nakonec musíme přidávat výjimky či úpravy.
- Abstrakce může mírně **snížit výkon** (např. přidáním dalších vrstev volání) a ztížit ladění, protože při řešení problémů musíme často pronikat skrz více úrovní abstrakce k detailům implementace.

Příklad (Python): Následující kód demonstruje abstrakci pomocí **abstraktní třídy** `Zvire`, která předepisuje metodu `vydej_zvuk()` pro potomky, ale neimplementuje ji. Třídy `Pes` a `Kocka` tuto abstrakci rozvíjejí – každá specifickým způsobem implementuje, jaký zvuk zvíře vydá:

```
from abc import ABC, abstractmethod

class Zvire(ABC):
    @abstractmethod
    def vydej_zvuk(self):
        pass

class Pes(Zvire):
    def vydej_zvuk(self):
        print("Haf haf!")

class Kocka(Zvire):
    def vydej_zvuk(self):
        print("Mňau!")

# Příklad použití:
zvirata = [Pes(), Kocka()]
for z in zvirata:
    z.vydej_zvuk()  # Výstup: Haf haf! a Mňau!
```

V abstraktní třídě `Zvire` není detailní implementace zvuku – pouze se tím říká, že **každé zvíře musí umět vydat zvuk**. Konkrétní třídy `Pes` a `Kocka` pak doplňují konkrétní implementace (štěknutí, mňouknutí). Vytváříme list `zvirata` obsahující instanci psa a kočky, a v cyklu voláme stejnou metodu `vydej_zvuk()` – díky abstrakci a **polymorfismu** (viz níže) každá instance reaguje podle své konkrétní třídy.

Zapouzdření

Definice: Zapouzdření (encapsulation) je princip, který zajišťuje **skrytí vnitřního stavu a implementace objektu** před vnějším okolím. Objekt vystavuje pouze rozhraní (veřejné metody), pomocí kterého s ním mohou ostatní komunikovat, a zabraňuje přímému přístupu k interním datům (Objektově orientované programování – Wikipedie). Jinými slovy, **jak** objekt uvnitř funguje a jaká data uchovává, je před ostatními objekty „ukryto“ – vnějšek využívá pouze to, co objekt poskytuje navenek. Například objekt "Auto" může mít interně různá data (stav paliva, teplotu motoru apod.), ale navenek poskytuje jen metody jako `nastartuj()`, `zastav()`, `zrychli()` atd., čímž brání tomu, aby jiné části

programu přímo manipulovaly s jeho vnitřnostmi (např. přímo měnily hodnotu paliva v nádrži).

Účel: Cílem zapouzdření je **ochrana a integrita dat** a snazší údržba kódu. Díky zapouzdření nemohou jiné objekty přistoupit k vnitřnímu stavu objektu způsobem, který by mohl vést k nekonzistenci nebo chybnému stavu (Objektově orientované programování – Wikipedie). Změny v interní implementaci objektu lze provést, aniž by to ovlivnilo kód, který objekt používá, pokud zachováme stejné rozhraní. To zlepšuje **modularitu** a **udržovatelnost** – můžeme upravovat a vylepšovat vnitřek objektu (např. optimalizovat algoritmy) a ostatní části systému to nijak nepoznají, protože nadále komunikují stejnými metodami.

Výhody zapouzdření:

- **Ochrana stavu objektu:** Jiné objekty nemohou neoprávněně měnit vnitřní data objektu. Tím se předchází chybám a logickým nekonzistencím (objekt si sám kontroluje, jaké změny do svého stavu pustí).
- **Jasně definované rozhraní:** Uživatel objektu vidí jen veřejné metody, takže **použití objektu je jednodušší** – není třeba rozumět detailům uvnitř. To podporuje *modulární návrh* a **znovupoužitelnost** kódu (objekt může být použit v různých kontextech, pokud poskytuje očekávané metody).

Nevýhody zapouzdření:

- **Více kódu navíc:** Pro přístup k interním datům často musíme psát tzv. *gettery a settery* (metody pro získání nebo nastavení hodnot) nebo používat jiné mechanismy (např. vlastnosti v Pythonu), což může znamenat více kódu oproti přímému přístupu k proměnným.
- **Potenciální ztráta flexibility:** Pokud potřebujeme skutečně získat přístup k nějakému internímu detailu objektu (např. pro ladění či specifické optimalizace), zapouzdření tomu brání. Řešením ale je upravit návrh rozhraní objektu tak, aby poskytovalo potřebné informace kontrolovaně (stále lepší než narušit zapouzdření).

Příklad (Python): V Pythonu se zapouzdření neprosazuje striktně pomocí klíčových slov (jako např. `private` v jiných jazycích), ale existuje konvence používat u *interních atributů* dvojité podtržítko na začátku názvu, což aktivuje tzv. *name mangling* (zamezení jednoduchého přístupu zvenčí). V následující třídě `BankovniUcet` je atribut zůstatku označen jako privátní (`__zustatek`) a lze s ním manipulovat jen přes metody `vloz` a `ziskej_zustatek`:

```
class BankovniUcet:
    def __init__(self, pocatecni_zustatek):
        self.__zustatek = pocatecni_zustatek  # "privátní" atribut

    def vloz(self, castka):
        if castka > 0:
            self.__zustatek += castka

    def ziskej_zustatek(self):
        return self.__zustatek

# Příklad použití:
ucet = BankovniUcet(100)
ucet.vloz(50)
print(ucet.ziskej_zustatek())  # Výstup: 150
# Pozn.: Přímý přístup k atributu __zustatek (např. ucet.__zustatek) není mo
žný.
```

V tomto příkladu **není možné** z vnějšku přímo přistoupit k proměnné `__zustatek` (pokud by to někdo zkusil, `AttributeError` chybu). Uživatel účtu musí použít veřejnou metodu `ziskej_zustatek()`. Třída tak **kontroluje platnost operací** – např. v metodě `vloz` můžeme ošetřit, že se vkládá jen kladná částka. Zapouzdření tedy zajišťuje, že objekt `ucet` **uchovává konzistentní stav** a nabízí bezpečnou sadu operací pro manipulaci s tímto stavem.

Dědičnost

Definice: Dědičnost umožňuje definovat **novou třídu na základě existující třídy**. Nová třída (tzv. *potomek* nebo *podtřída*) **zdědí** všechny atributy a metody původní třídy (tzv. *předka* nebo *rodičovské třídy*) a může je **rozšířit nebo pozměnit**. Dědičnost vytváří hierarchii tříd – objekt podtřídy je zároveň specifickým druhem objektu nadřazené třídy. Například v programu můžeme mít obecnou třídu *Člověk* a z ní odvozenou třídu *Svářeč*; *Svářeč* bude automaticky mít vše, co má *Člověk* (jméno, schopnost chodit atd.), a navíc může přidat něco specifického (například metodu *svařuj*) (Objektově orientované programování – Wikipedie).

Účel: Dědičnost podporuje **znovupoužití kódu** a usnadňuje vytváření specializovaných typů. Společné vlastnosti a funkce lze umístit do základní

třídy, takže je není nutné znovu implementovat v každé podobné třídě – potomci je zdědí. Zároveň umožňuje vytvářet **logickou hierarchii** tříd, která odpovídá reálným vztahům *"je to druh"* – např. pes *je* druh zvířete, auto *je* druh vozidla apod. To napomáhá organizaci programu: obecné chování je v předkovi, specifické v potomcích.

Výhody dědičnosti:

- **Znovupoužití a údržba kódu:** Sdílené funkčnosti jsou implementovány jen jednou (v předkovi) a využívá je více tříd. Opravy nebo vylepšení v základní třídě se projeví i u potomků, aniž by bylo nutné zasahovat do jejich kódu (Objektově orientované programování – Wikipedie).
- **Přehlednost a logika modelu:** Třídy lze uspořádat do hierarchické struktury, která často odráží přirozené vztahy. Program může být **přehlednější**, protože podobné objekty sdílejí základ a liší se jen v detailu.

Nevýhody dědičnosti:

- **Zvýšená provázanost:** Potomek je pevně svázán s implementací předka. Změna v kódu předka může neúmyslně ovlivnit (nebo narušit) funkčnost potomků. Je tedy nutné pečlivě navrhovat rozhraní předků, aby se minimalizovaly negativní dopady změn.
- **Komplikace při nesprávném použití:** Pokud je hierarchie tříd příliš hluboká nebo nepřehledná, může být program obtížně pochopitelný a udržovatelný. Někdy se dědičnost používá i tam, kde by byla vhodnější jiná vazba (např. kompozice objektů) – nevhodná dědičnost pak vede k nehodícím se vztahům typu *"je to druh"*.

Příklad (Python): Následující ukázka definuje třídu `Clovek` a její podtřídu `Svarec`. **Svářeč** dědí od **Člověka** atribut `jmeno` a metodu `pozdrav` a přidává si vlastní specializované chování (překrývá metodu `pozdrav` a doplňuje novou metodu `svaruj`):

```
class Clovek:
    def __init__(self, jmeno):
        self.jmeno = jmeno
    def pozdrav(self):
        print(f"Ahoj, jsem {self.jmeno}.")

class Svarec(Clovek):
    def pozdrav(self):
```

```
        print(f"Jsem svářeč {self.jmeno}.")
    def svaruj(self):
        print(f"{self.jmeno} svařuje kov.")

# Příklad použití:
osoba = Svarec("Pavel")
osoba.pozdrav()  # Výstup: "Jsem svářeč Pavel."
osoba.svaruj()   # Výstup: "Pavel svařuje kov."
```

Třída `Svarec` využívá všeobecné vlastnosti definované ve třídě `Clovek` a rozšiřuje je. Objekt `osoba` vytvořený jako `Svarec` má **všechny atributy a metody člověka** (jméno, `pozdrav()`) a navíc metodu `svaruj()`. Zároveň jsme v `Svarec.pozdrav()` ukázali **polymorfní chování** – svářeč má vlastní verzi metody `pozdrav`, která *překrývá* (override) metodu zděděnou. Díky tomu volání `osoba.pozdrav()` použije **odlišnou implementaci** než u obecného člověka. Dědičnost tak umožňuje vytvářet specializované objekty, které se z velké části chovají jako obecný *Clovek*, ale v něčem se liší.

Polymorfismus

Definice: Polymorfismus (mnohotvárnost) znamená, že **stejná operace může mít různou formu chování** v závislosti na tom, nad jakým objektem je vykonávána. V praxi to znamená, že pokud různé třídy definují *stejně pojmenovanou metodu*, lze na jejich instance volat tuto metodu **stejným způsobem**, ale každá třída na ni může reagovat odlišně. Polymorfismus je úzce spjat s dědičností: objekty odvozené od stejného předka mohou *překrývat* jeho metody a reagovat tak po svém. Díky polymorfismu můžeme s různými objekty zacházet jednotně, pokud sdílejí společné rozhraní (Objektově orientované programování – Wikipedie).

Existují dva hlavní typy polymorfismu:

- **Polymorfismus podmíněný dědičností:** Je přítomný, když různé podtřídy jednoho předka implementují (překryjí) zděděnou metodu různě. Potom platí, že všude tam, kde program očekává instanci základní třídy, lze použít instanci jakékoli její podtřídy (Objektově orientované programování – Wikipedie). Například funkce napsaná tak, aby pracovala s objektem typu `Clovek`, bude správně fungovat i pro objekty typu `Svarec` (potomek `Clovek`), přičemž může volat polymorfně překrytou metodu `pozdrav()` konkrétního svářeče.

- **Polymorfismus nepodmíněný dědičností (tzv. *duck typing*):** Typický pro dynamické jazyky jako Python. Není nutná společná nadtřída; postačí, když různé třídy definují stejné metody. Pokud dvě třídy mají metodu stejného jména a signatury (např. `obsah()` u kruhu i obdélníku), můžeme na obě volat `objekt.obsah()` a získat smysluplný výsledek. V Pythonu se říká *„pokud to chodí jako kachna a kváká to jako kachna, pak je to kachna“* – nezáleží na třídní hierarchii, ale na dostupnosti požadovaných metod.

Účel: Polymorfismus zvyšuje **flexibilitu a rozšiřitelnost kódu**. Umožňuje psát obecný kód nezávislý na konkrétních třídách – např. funkci, která volá `objekt.vykresli()`, můžeme použít pro vykreslení libovolného tvaru (kružnice, čtverce, trojúhelníku…), pokud všechny tyto objekty mají metodu `vykresli`. Při přidání nového tvaru (třídy) nemusíme měnit kód funkce ani ostatních částí systému, stačí zajistit, že nová třída obsahuje požadovanou metodu. Polymorfismus tedy podporuje **modularitu** (oddělení kódu) a **snadné rozšiřování** programů.

Výhody polymorfismu:

- **Jednotné rozhraní:** Můžeme k různým objektům přistupovat stejným způsobem. To zjednodušuje návrh rozhraní funkcí a metod – není třeba psát duplicity nebo velké podmínky pro různé typy.
- **Snadné přidávání nových typů:** Nová třída implementující existující rozhraní (metody) může být začleněna, aniž by bylo nutné upravovat stávající kód. Stačí, že nový objekt reaguje na požadované zprávy/metody.

Nevýhody polymorfismu:

- **Ztížené sledování kódu:** Při čtení programu může být méně zřejmé, **která konkrétní implementace** metody se zavolá – záleží to na typu objektu v daném čase. To může zkomplikovat ladění, zvláště v hierarchiích s mnoha úrovněmi nebo u dynamických typů.
- **Potenciální runtime chyby:** V dynamicky typovaných jazycích (Python) polymorfismus spoléhá na to, že objekt má požadovanou metodu. Pokud se pokusíme volat metodu, kterou objekt nemá (chybný předpoklad typu), program havaruje až za běhu. Je tedy potřeba pečlivě testovat, případně používat kontrolu pomocí funkcí `hasattr` apod.

Příklad (Python): Ukázka polymorfismu pomocí **duck typing**. Definujeme dvě nesouvisející třídy `Obdelnik` a `Kruh`, z nichž každá má metodu `obsah()`. Dále napíšeme funkci `vypis_obsah()`, která pro libovolný objekt zavolá jeho metodu

`obsah()` a vypíše výsledek. Díky polymorfismu můžeme tu stejnou funkci použít pro obdélník i kruh (a případně jakýkoli jiný objekt, který má metodu `obsah`):

```
class Obdelnik:
    def __init__(self, sirka, vyska):
        self.sirka = sirka
        self.vyska = vyska
    def obsah(self):
        return self.sirka * self.vyska

class Kruh:
    def __init__(self, polomer):
        self.polomer = polomer
    def obsah(self):
        import math
        return math.pi * (self.polomer ** 2)

# Funkce, která vypíše obsah libovolného objektu s metodou obsah():
def vypis_obsah(objekt):
    print(f"Obsah je {objekt.obsah():.2f}")

# Příklad použití:
tvary = [Obdelnik(3, 4), Kruh(5)]
for tvar in tvary:
    vypis_obsah(tvar)
# Výstup:
# Obsah je 12.00
# Obsah je 78.54
```

Třídy `Obdelnik` i `Kruh` poskytují stejnou metodu `obsah()`, i když nejsou ve vztahu předek-potomek. Funkce `vypis_obsah` je napsána obecně – nezajímá se o typ objektu, volá prostě `objekt.obsah()`. V seznamu `tvary` pak můžeme mít různé objekty a pro každý z nich funkci zavolat. Každý objekt **polymorfně** použije svou vlastní implementaci metody `obsah()` (obdélník počítá obsah jako šířka×výška, kruh jako πr^2). Díky polymorfismu tak lze psát kód, který **automaticky přizpůsobí své chování typu objektu**, aniž by to bylo v kódu explicitně ošetřeno podmínkami.

Třída

Definice: Třída je **šablona (nebo také datový typ) pro vytváření objektů**. Definuje, **jaké atributy a metody** budou objekty dané třídy mít. Lze si ji představit jako popis nebo předpis objektu – například třída `Auto` může popisovat, že *každé auto má* určitý objem nádrže, barvu karoserie a metody jako nastartuj/zastav, apod. Konkrétní auto v programu pak bude instance této třídy. Formálně se dá říct, že třída je abstrakce množiny podobných objektů – **předpis, jak vyrobit objekt daného typu** (Objektově orientované programování – Wikipedie). Třídy v kódu také představují **logické uspořádání** kódu: uvnitř třídy definujeme související funkčnost na jednom místě.

Účel: Třídy umožňují **organizovat program** do logických celků (objektů) a **vícenásobně využívat kód**. Když jednou definujeme třídu, můžeme vytvořit libovolný počet objektů této třídy a všechny automaticky mají chování podle definice. Třída tedy slouží jako *návod* – například třída `Kniha` může definovat, že každá kniha má název, autora a metodu pro vypsání informací. Vytvořením tří různých instancí třídy `Kniha` získáme tři konkrétní knihy s různými daty, ale stejnou strukturou a funkcemi. Bez tříd bychom museli pro každou knihu zvlášť udržovat její data a funkce, třída nám poskytne **jednotný rámec**.

Výhody třídy:

- **Znovupoužitelnost a úspora kódu:** Jednou definovaná třída umožňuje vytvořit mnoho objektů, aniž bychom pro každý psali vše od nuly. Sdílená implementace v třídě snižuje duplicitní kód.
- **Přehlednost a údržba:** Kód je strukturován podle reálných entit – snadněji se v něm orientuje a mění se. Pokud potřebujeme něco změnit na všech autech, upravíme třídu `Auto` (namísto úprav u každého objektu zvlášť). Třídy také umožňují využití principů OOP (abstrakce, zapouzdření, dědičnost, polymorfismus) k psaní čistšího a modulárního kódu.

Nevýhody třídy:

- **Režie navíc pro jednoduché úlohy:** Pro velmi jednoduché skripty může být použití tříd zbytečně složité – definování třídy a vytváření objektů může být „overhead“, když by stačilo např. použít jednu funkci nebo jednoduchou datovou strukturu.
- **Nutnost porozumění OOP konceptům:** Začátečník se musí naučit syntaxi a filozofii tříd a objektů. Špatně navržená třída pak může způsobit více problémů než užitku (např. pokud má nevhodně nastavené zapouzdření

nebo nejasnou odpovědnost). Jde však spíše o nevýhodu spojenou s učením a návrhem než o technický nedostatek tříd.

Příklad (Python): Třída `Kniha` níže popisuje model knihy s atributy `nazev` a `autor` a metodou `info()`. Podle této třídy vytvoříme instanci (objekt) konkrétní knihy a zavoláme metodu `info`:

```
class Kniha:
    def __init__(self, nazev, autor):
        self.nazev = nazev
        self.autor = autor

    def info(self):
        print(f"\"{self.nazev}\", autor: {self.autor}")

# Vytvoření instance:
kniha1 = Kniha("Malý princ", "Antoine de Saint-Exupéry")
kniha1.info()  # Výstup: "Malý princ", autor: Antoine de Saint-Exupéry
```

Třída `Kniha` funguje jako **šablona** – říká, že *každá* kniha má název a autora a umí se sama popsat. Metoda `__init__` je konstruktor, který se volá při vytváření objektu (`kniha1 = Kniha(...)`) a zajistí nastavení atributů. Instanci `kniha1` pak můžeme používat: nese v sobě data *"Malý princ"/"Antoine de Saint-Exupéry"* a sdílí s ostatními knihami (instancemi `Kniha`) definici metody `info()`. Kdybychom chtěli přidat další vlastnost (např. rok vydání) nebo funkci (např. porovnat dvě knihy), upravíme definici třídy – to se projeví u všech instancí. **Třída tedy slouží jako centrální místo, kde je definováno, co objekty umí.**

Instance (objekt)

Definice: Instance je **konkrétní objekt vytvořený podle třídy**. Pokud je třída šablonou nebo předpisem, pak instance je výsledný "produkt" podle tohoto předpisu v běžící aplikaci. Říkáme, že objekt je *instancí* určité třídy, což znamená, že má strukturu a chování definované touto třídou (Základní pojmy objektově orientovaného programování - Builder.cz - Informacni server o programovani). Například můžeme mít třídu `Auto` a z ní vytvořit instanci `auto1` reprezentující konkrétní auto – tato instance má vlastní konkrétní hodnoty atributů (značka, barva atd.), ale sdílí definici se všemi ostatními auty vytvořenými z téže třídy.

Účel: Pomocí instancí můžeme **modelovat reálné jednotliviny** v programu. Třída sama o sobě je jen abstraktní popis; teprve vytvořením instance získáme **konkrétní objekt**, se kterým můžeme pracovat (volat metody, číst a měnit jeho atributy). Můžeme vytvářet libovolný počet instancí z jedné třídy – tím OOP umožňuje jednoduše reprezentovat více podobných entit bez duplicitního kódu. Každá instance má **vlastní kopii atributů**, takže uchovává svůj vlastní stav. Například třída `Pes` může sloužit jako předloha a každé volání `Pes("Azor")` vytvoří nového psa se jménem Azor. Pokud vytvoříme další instanci `Pes("Buddy")`, dostaneme nezávislý objekt s jiným jménem. To je zásadní pro modelování více objektů současně.

Výhody instance:

- **Množství nezávislých objektů:** Z jedné třídy lze vytvořit mnoho instancí. Každá instance představuje samostatný objekt – můžeme tak v programu snadno pracovat s více "entitami" najednou (např. stovky různých uživatelských objektů typu `Uzivatel`).
- **Oddělený stav:** Každá instance má vlastní stav (hodnoty atributů), který je nezávislý na ostatních instancích. Změna atributu u jednoho objektu neovlivní atributy jiného objektu, i když patří do stejné třídy. To umožňuje simulovat reálnou situaci, kde každý konkrétní "jedinec" má své údaje.

Nevýhody instance:

- **Spotřeba zdrojů pro velký počet objektů:** Každá instance zabírá určité místo v paměti. Pokud program vytvoří obrovské množství instancí, může to vést k vyšší spotřebě paměti a případně ke snížení výkonu při jejich zpracování.
- **Složitost správy mnoha objektů:** Když pracujeme s mnoha instancemi, může být náročnější sledovat všechny jejich stavy a interakce. Je potřeba mít dobře navržené struktury, aby se s kolekcemi objektů (např. seznamem všech aut v simulaci) pracovalo efektivně a přehledně.

(Poznámka: Samotný koncept "instance" nemá přímo nevýhody ve smyslu špatné vlastnosti – výše zmíněné body jsou spíše obecné praktické aspekty práce s objekty. Instancím se v OOP nelze vyhnout, protože bez instancí by třídy byly jen teorie.)

Příklad (Python): Uvažujme třídu `Auto` definovanou výše. Vytvoříme dvě různé instance této třídy a ukážeme, že každá má svůj vlastní stav (značku). Změna atributu u jedné instance neovlivní druhou instanci:

```
class Auto:
    def __init__(self, znacka):
        self.znacka = znacka

# Vytvoření dvou instancí (objektů) třídy Auto:
auto1 = Auto("Škoda")
auto2 = Auto("Ford")

print(auto1.znacka)  # Výstup: Škoda
print(auto2.znacka)  # Výstup: Ford

# Změna stavu první instance:
auto1.znacka = "Toyota"
print(auto1.znacka)  # Výstup: Toyota
print(auto2.znacka)  # Výstup: Ford
```

Nejprve vytvoříme `auto1` a `auto2`. Oba objekty jsou typu `Auto`, ale každý má jinou hodnotu atributu `znacka`. Po vytvoření vypíšeme `auto1.znacka` (`"Škoda"`) a `auto2.znacka` (`"Ford"`). Následně změníme značku u `auto1` na `"Toyota"` a znovu vypíšeme obě značky. Výsledek ukazuje, že `auto2` **zůstalo nezměněno** – každý objekt si uchovává svůj vlastní stav. Tím se potvrzuje, že instance stejné třídy fungují **nezávisle**: úpravy jednoho objektu nemají vedlejší účinky na jiný objekt, pokud to explicitně neprogramujeme.

Každá instance existuje na určité adrese v paměti a v Pythonu s nimi pracujeme prostřednictvím **referencí** (proměnných jako `auto1`, `auto2`, které odkazují na tyto objekty). Když instance již nejsou potřeba, Python je automaticky uvolní z paměti (garbage collection).

Závěr

V objektově orientovaném programování spolu výše popsané pojmy úzce souvisejí. **Třídy** definují strukturu a chování, **instance (objekty)** jsou konkrétní realizace tříd. Pomocí **abstrakce** určujeme podstatné vlastnosti objektů a ignorujeme detaily, **zapouzdření** zabezpečuje jejich vnitřní stav, **dědičnost** umožňuje vytvářet hierarchie tříd a přebírat funkcionalitu, a **polymorfismus** zajišťuje, že objekty mohou být použity záměnně, i když se v detailech chovají odlišně. Tyto principy společně usnadňují tvorbu složitých software tím, že kód je strukturovaný, modulární a lépe odpovídá reálným modelům světa. Při

dobrém zvládnutí OOP dokáže programátor efektivně modelovat problémovou doménu a vytvářet udržovatelný a rozšiřitelný kód. (Objektově orientované programování – Wikipedie) (Základní pojmy objektově orientovaného programování - Builder.cz - Informacni server o programovani)