

# Principy objektově orientovaného programování

## Vztah k ostatním paradigmatům

Každé programovací paradigma se snaží vyřešit problém sdíleného stavu programu.

- imperativní programování - výchozí způsob - stav je libovolně měnitelný a je sdílen v globálních a lokálních proměnných
- funkcionální programování - „žádný sdílený stav neexistuje“ - stav je neměnný, mezi tzv. „pure“ funkcemi bez vedlejších účinků se posílají kopie stavu
- objektové programování - stav je rozdělen na malé části a zapouzdřen logikou programu v objektech

To znamená, že konkrétní hodnoty jsou uloženy jako atributy těchto objektů. Zpracování atributů spolu s celkovou komunikací s objekty probíhá pomocí kódu, který je obsažený v metodách objektů. Navenek pak program působí jako několik navzájem spolupracujících objektů, což umožňuje snadnější přenos kódu mezi různými projekty a jednodušší úpravu již existujícího kódu pomocí dědičnosti. Další výhodou je také větší zabezpečení dat díky viditelnosti atributů.

Nevýhodou objektově orientovaného programování je jeho větší náročnost na paměť a výpočetní rychlost, proto se používá hlavně u moderních počítačů, kde jsou tyto nevýhody téměř nezatelné. U mikropočítačů a jednoduchých jednoúčelových strojů je však výhodnější použití strukturovaného programování.

## Zapouzdření

Zapouzdření není výhradně koncept OOP. Obecně v programování znamená, že nějaké data nebo logiku schováme za restriktivním rozhraním.

V OOP je tento koncept implementován třídami, kde data jsou schována v podobě privátních atributů, které můžeme manipulovat pomocí public metod. Příkladem zapouzdření je například kolekce HashMap v jazyce Java (skrývá implementaci hašovací tabulky za metodami `.put()` nebo `.getOrDefault()`).

## Dědičnost

Dědičnost umožňuje tvorbu nových tříd podle již vytvořených tříd. Tyto odvozené třídy sdělí všechny public a protected atributy a metody rodičovských tříd. Odvozené třídy si mohou tato data různě upravovat nebo přidávat další.

např:

```
class Zpevak extends Clovek {  
  
    zpivej(){};  
  
    tancuj(){}; }  
}
```

V tomto příkladu jsme vytvořili novou třídu *Zpevak*, která je odvozena od třídy *Clovek*. Objekt třídy *Zpevak* tedy může využívat všechny atributy a metody třídy *Clovek* (např. metoda *vstan()* a atribut *barvaOci*) a zároveň nově vytvořené metody *zpivej()* a *tancuj()*.

Podle kritiků by se dědičnost měla používat velmi vzácně, protože vytváří problém přílišné abstrakce a hluboké hierarchie dědičnosti, ve které člověk nepozná, kde je logika skutečně implementována. Z těchto důvodů existuje nepsané pravidlo, že dědičnost by měla být hluboká pouze 1 vrstvou. (dceřiná třída nesmí být rodičovskou třídou pro jiné třídy)

Na druhou stranu existují případy, kde je vícevrstvá dědičnost velmi výhodná - například při implementaci datových struktur ve standardní knihovně jazyku Java nebo ve vývoji her (hierarchie typu *Entity* → *Character* → *Player*).

## Vícenásobná dědičnost

Počet rodičovských tříd je určen použitým programovacím jazykem (např. Java umožňuje pouze jednu rodičovskou třídu, C++ umožňuje více).

## Polymorfismus

Umožňuje použít jednotné rozhraní pro práci s různými typy objektů např:

instance třídy *mladyClovek* bude metodu *presunSe()*; vykonávat za pomoci metody *chod*, ale instance třídy *staryClovek* bude tu samou metodu *presunSe()*; vykonávat za pomoci *chodOHoli*

To znamená, že i když každý objekt tuto metodu vykonává jinak, z vnějšího hlediska se tváří stejně a my nemusíme tedy přemýšlet, jak přesně toho u různých objektů docílit.

Pod pojmem polymorfismus můžeme také rozumět **Přetěžování metod** - to znamená že metoda může fungovat více různými způsoby, které se rozliší podle druhu a počtu parametrů. Např metoda *rekni(string vyrok)*; umožní objektu *clovek* říct nějaký výrok jen tak do prázdna, zatímco metoda *rekni(string vyrok; clovek prijemce; zpusobHlasitosti hlasitost)* umožní tomu samému objektu říct výrok konkrétnímu příjemci a zvolenou hlasitostí.

V některých programovacích jazycích umožňuje polymorfismus metodám také různé zpracování parametrů různého typu.

## Viditelnost atributů a metod

Nastavením viditelnosti dat můžeme určit, které části programu budou mít k těmto datům přístup. Nastavit můžeme tři základní možnosti.

- `private` - data jsou viditelná pouze pro konkrétní objekt
- `public` - data jsou viditelná komukoli
- `protected` - data jsou viditelná pouze pro konkrétní třídu a odvozené třídy

## Rozhraní

Zjistíme, že programátor umí psát na počítači a účetní také. Intuitivně cítíme, že nebudou mít mnoho dalších společných schopností a navíc tuto dovednost může mít napříč povoláními leckdo, proto nemá smysl tvořit třídu na způsob `ČlovekPracujícíSPOčítačem` a od ní dědit `Programátora` a `Účetní`, ale je výhodnější například vytvořit rozhraní `SchopenPsátNaPočítači` s požadavkem na metodu `napišNaPočítači()` a upravit třídy `Programátor` a `Účetní` tak, aby toto rozhraní implementovaly, tedy předepsanou metodu, a to každý po svém. V definici rozhraní nemůže být obsažen kód (implementace) dané metody, ale všechny třídy, které toto rozhraní implementují, musí být schopny se s příkazem `napišNaPočítači()` nějak vypořádat.

Obdobně může abstraktní třída předepisovat doimplementování metod, pro které ona sama nemá vlastní kód, ale jen předpis abstraktní metody.

From:

<http://wiki.gml.cz/> - GMLWiki

Permanent link:

<http://wiki.gml.cz/informatika:maturita:19a?rev=1771519956>

Last update: **19. 02. 2026, 17.52**

