OOP hodně zjednodušený úvod

Motto

Nevynalézat kolo...

Patrně si říkáte, proč se učit algoritmy, vždyť to je zbytečná práce... a budete mít do určité míry pravdu. Opravdu nemá smysl znovu vynalézat kolo, protože jeho existující implementace jsou často perfektní a hlavně prověřené. Na druhou stranu je nutné vědět, jakým způsobem jsou vytvořeny základy staveb, které používáme.

...ale znát jeho principy

Objektově orientovaný přístup

Jedná se o filozofii a způsob myšlení, designu a implementace, kde klademe důraz na znovupoužitelnost.

Přístup nalézá inspiraci v průmyslové revoluci - vynález základních komponent, které budeme dále využívat (např. když stavíme dům, nebudeme si pálit cihly a soustružit šroubky, prostě je již máme hotové).

Přednosti OOP

Poskládání programu z komponent je výhodnější a levnější.

Můžeme mu věřit, je otestovaný (o komponentách se ví, že fungují, jsou otestovány a udržovány).

Pokud je někde chyba, stačí ji opravit na jednom místě.

Jsme motivováni k psaní kódu přehledně a dobře, protože ho po nás používají druzí nebo my sami v dalších projektech.

Poznámka: člověk je od přírody líný a kdyby nevěděl, že se jeho kód bude znovu využívat, nesnažil by se ho psát kvalitně ☺

Princip OOP

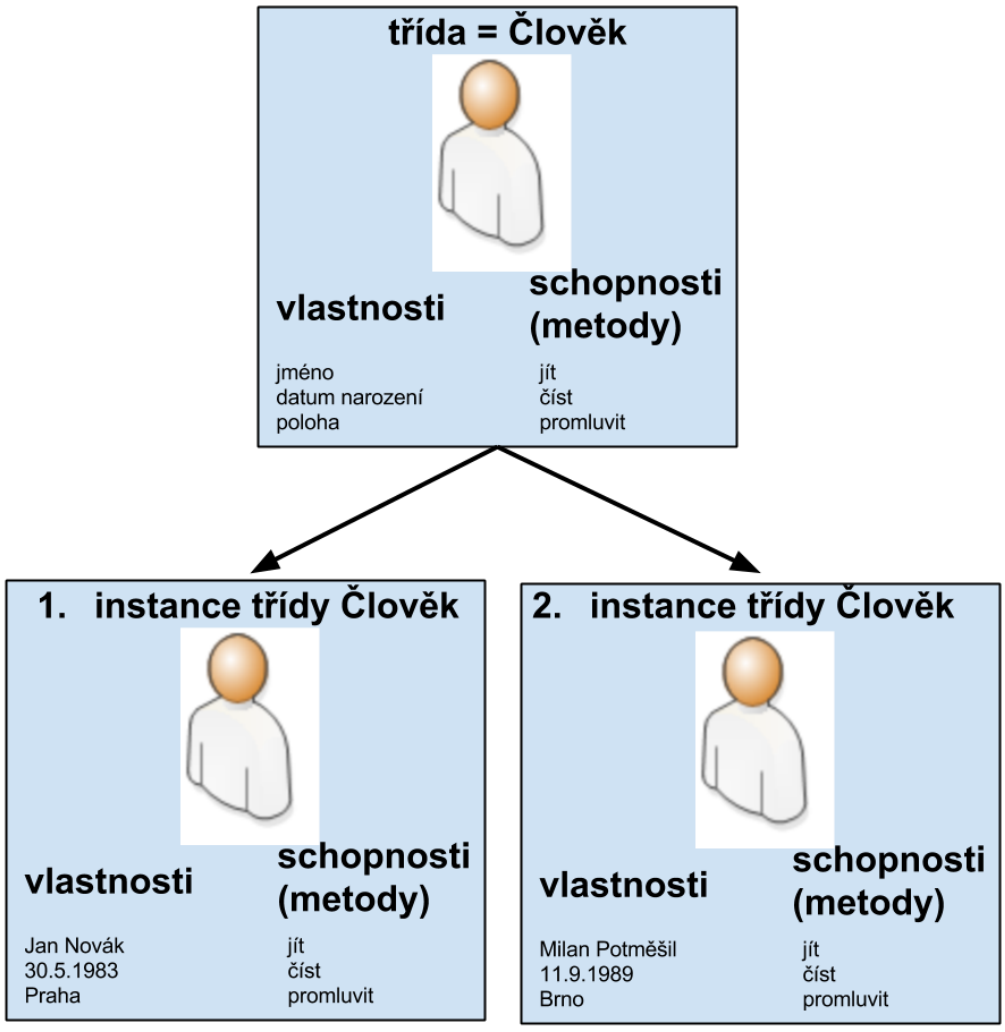
Snažíme se nasimulovat realitu tak, jak ji jsme zvyklí vnímat. Můžeme tedy říci, že se odpoutáváme od toho, jak program vidí počítač (stroj) a píšeme program spíše z pohledu programátora (člověka

Jako jsme nahradili assembler pro člověka čitelnými matematickými zápisy, nyní jdeme ještě dál a nahradíme i ty.

Jde tedy o určitou úroveň abstrakce nad programem.

To má značné výhody už jen v tom, že je to pro nás přirozenější a přehlednější.

Objekt, třída, instance

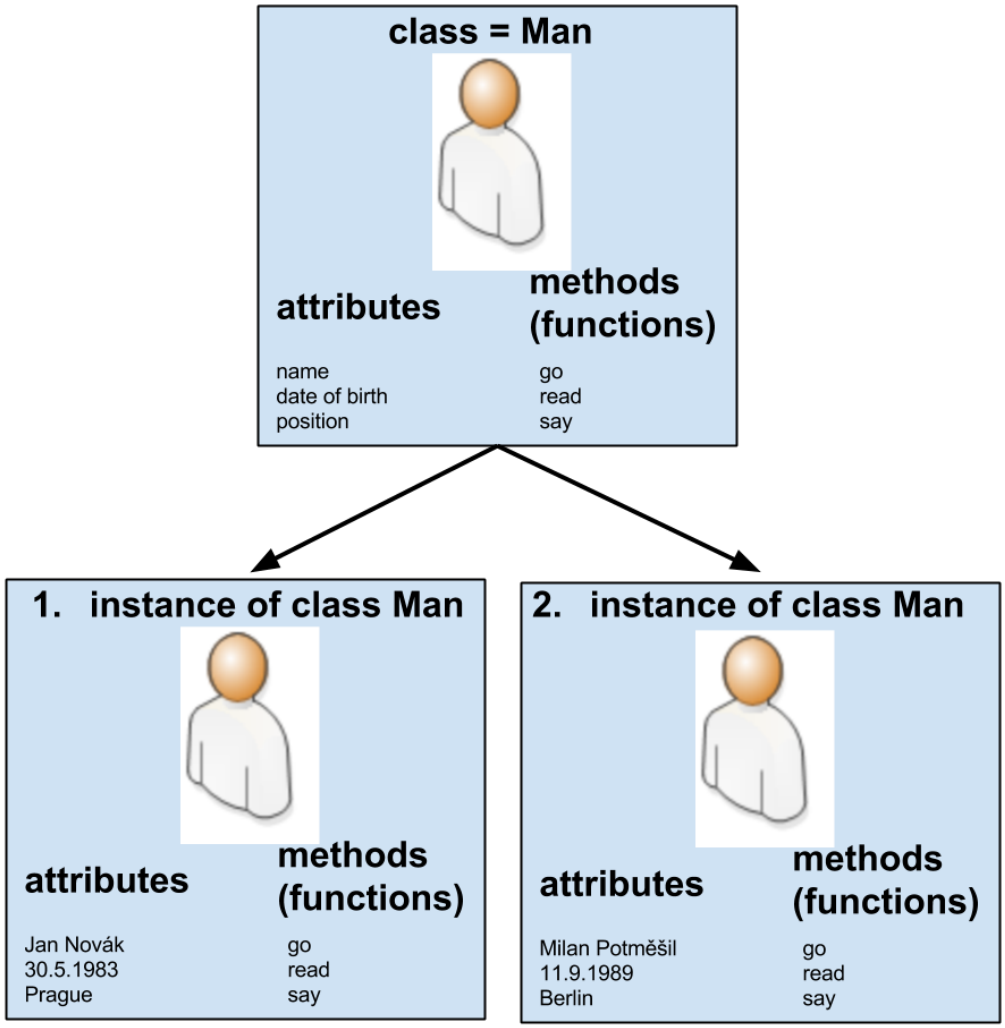


Mějme nějaký pojem z reálného světa, třeba „člověk“ nebo „databáze“. Tento pojem můžeme chápat ve dvou významech:

1. jako abstrakci, která shrnuje všechny společné znaky pojmu. Příkladem může být „člověk“, který má dvě ruce, dvě nohy, hlavu a tělo, někdy též rozum a podobně. Tomuto významu říkáme **třída**; mluvíme o třídě Člověk, o třídě Databáze apod. Třída určuje obecně, které vlastnosti a schopnosti bude mít.
2. jako konkrétního člověka; v tomto významu mluvíme o **instanci třídy** nebo prostě o **objektu**. Každý objekt má své konkrétní vlastnosti i své konkrétní metody.

Můžeme si představovat, že třída je „formička“, podle které vyrábíme různé instance (=objekty).

Třídy i objekty mají své **atributy** a **metody**.



Atributy

Atributy objektu jsou vlastnosti neboli data, která uchovává (např. u člověka jméno a věk, u databáze heslo). Jedná se o prosté proměnné, se kterými jsme již stokrát pracovali. Někdy o nich hovoříme jako o vnitřním stavu objektu.

Objekty (instance) mají tytéž atributy jako má jejich třída. Atributy objektů mohou mít své individuální, pro každý objekt jiné, **hodnoty**.

Metody

Metody jsou schopnostmi, které umí objekt vykonávat. U člověka by to mohly být metody: **jdi()**, **čti()** nebo **řekni()**.

Z programátorského pohledu to vlastně jsou funkce.

Ve starších jazycích metody nepatřily objektům, ale volně se nacházely v modulech.

Nevýhodou je samozřejmě zejména to, že metoda zde nikam nepatří. Není způsob, jakým si vyvolat seznam toho, co se s objektem dá dělat a v kódu je nepořádek.

Navíc nemůžeme mít 2 metody se stejným názvem, zatímco v OOP můžeme mít **uzivatel.vymaz()** a **clanek.vymaz()**.

Třída

Třída je vzor, je to šablona, podle kterého se objekty vytvářejí. Definuje jejich vlastnosti a schopnosti.

Objekt, který se vytvoří podle třídy, se nazývá **instance**.

Instance mají stejné rozhraní jako třída, podle které se vytváří, ale navzájem se liší svými daty (atributy).

Příklad: Třída x Instance

Mějme například třídu `Člověk` a od ní si vytvořme instance Jan Novák a Milan Potměšil.

Obě instance mají jistě ty samé metody a atributy, jako třída (např. jmeno a datum_narozeni) a metody (**jdi()** a **řekni()**), ale hodnoty v nich se liší (první instance má v atributu jméno hodnotu "Jan Novák" a datum narození 30.5.1983, druhá "Milan Potměšil" a 11.9.1989).

Základní vlastnosti OOP

zapouzdřenost

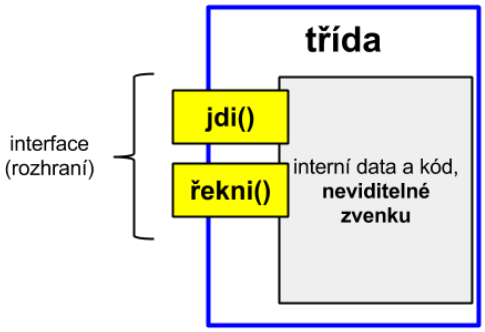
dědičnost

polymorfismus

události

Zapouzdření

Zapouzdření umožňuje skrýt některé metody a atributy tak, aby zůstaly použitelné jen pro třídu zevnitř. Objekt si můžeme představit jako **černou skříňku** (anglicky *blackbox*), která má určité rozhraní (interface), přes které jí předáváme instrukce/data a ona je zpracovává.



Nevíme, jak to uvnitř funguje, ale víme, jak se navenek chová a používá. Nemůžeme tedy způsobit nějakou chybu, protože využíváme a vidíme jen to, co tvůrce třídy zpřístupnil.

Příklad

Příkladem může být třída Člověk, která bude mít atribut `datumNarozeni` a na jeho základě další atributy: `plnolety` a `vek`.

Kdyby někdo objektu zvenčí změnil `datumNarozeni`, přestaly by platit proměnné `plnolety` a `vek`. Říkáme, že vnitřní stav objektu by byl nekonzistentní.

Toto se nám ve strukturovaném programování může klidně stát.

V OOP však objekt zapouzdříme a atribut `datumNarozeni` označíme jako privátní, zvenčí tedy nebude viditelný.

Naopak, ven vystavíme metodu `zmenDatumNarozeni()`, která dosadí nové datum narození do proměnné `datumNarozeni` a zároveň provede potřebný přepočet věku a přehodnocení plnoletosti.

Použití objektu je bezpečné a aplikace stabilní.

Zapouzdření tedy donutí programátory používat objekt jen tím správným způsobem.

Rozhraní (interface) rozdělí třídu na části veřejně přístupné (public) a vnitřní strukturu (private).

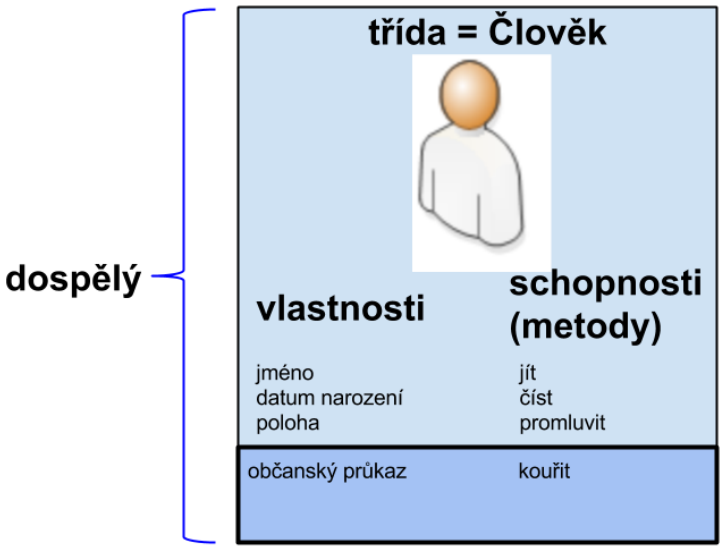
Dědičnost

Předpokládejme, že máme obecnou kategorii (*třídu*) Člověk, která nám popisuje obecné vlastnosti člověka.

Nyní dostaneme za úkol napsat jinou třídu, která bude popisovat dospělého člověka, třídu Dospělý. Na první pohled vidíme, že třídy **Člověk** a **Dospělý** se navzájem příliš neliší; hlavní rozdíl je v tom, že Dospělý má občanský průkaz a smí kouřit.

Samozřejmě bychom třídu Dospělý dokázali napsat znova, od začátku. Ale mělo by to podstatné nevýhody:

- jsme na to příliš líní
- celé psaní kódu, testování, vytváření dokumentace atd. bychom museli udělat znovu
- kód by se špatně udržoval, protože jakákoliv změna by se musela dělat současně ve třídě Člověk i ve třídě Dospělý.
-



Mnohem jednodušší je, vzít už hotovou třídu Člověk a jen ji trochu rozšířit. V našem příkladu – přidat k ní údaje o občanském průkazu a funkci *kouřit()*.

Tento způsob odvozování potomků od svých rodičů se nazývá **dědění**.

Pro dědění existují tři zajímavá pravidla.

Nelze ubírat

Metody a funkce lze jen přidávat nebo měnit, nelze je ubírat. Rodičovskou třídu už totiž někdo mohl použít jinde – kdybychom ji dodatečně modifikovali, mohli bychom vytvořit chybu.

V našem případě jsme při dědění Člověk → Dospělý mohli přidat i další funkce, třeba *řídit_auto()* nebo *pít_alkohol()*. Samozřejmě jsme také mohli přidat libovolné množství atributů.

Metody ani atributy, jakmile už jednou ve třídě jsou, nejde je při dědění vyndat. Jestliže třída Člověk má metodu *promluvit()*, tak už žádným zděděním nemůžeme dosáhnout toho, aby některá zděděná třída takovou metodu neměla.

Nicméně, u metody můžeme změnit význam, obsah zděděné metody. Například metodu *promluvit()* můžeme ve třídě Dospělý změnit tak, aby dělala něco úplně jiného než ve třídě Člověk. Také z jedné třídy (z Člověk) můžeme zdědit několik různých tříd Dospělý1, Dospělý2, … atd a každá z nich může *proluvit()* jinak.

Poznámka 1: Předefinovat můžeme jen metody, u atributů to nejde.

Poznámka 2: Metodu sice nemůžeme odstranit, ale při dědění ji můžeme změnit tak, aby nedělala nic. Ve výsledku to je (skoro) totéž.

Modifikovaná typová kontrola

O typech a typové kontrole budeme podrobně hovořit později. Nyní jen krátké vysvětlení.

Data jsou různých typů, například čísla a řetězce (texty). S každým typem dat se v počítači nakládá jinak. Například čísla lze násobit (řetězce nikoliv), ale řetězce zase lze rozdělit na slova (to u čísel nejde).

Proto je přirozené, každý datový typ zpracovávat odděleně a přísně kontrolovat, abychom někde typy nepřehodili, nezaměnili.

Ale co u dědění? Přísně vzato, třída Člověk je jiného typu než třída Dospělý. Jenže kdybychom nedovolili, aby se s Dospělým pracovalo jako s Člověkem, nemohli bychom například zjistit jeho jméno:<EN>< But what about inheritance? Strictly speaking, the class Man is of a different type than class Adults. But if we were not allowed to work with Adults as if they were of class Man, we could not find out his name, for example:/EN>

```
Dospělý.jmeno
```

by nešlo napsat, protože by šlo jedině

```
Člověk.jméno
```

To by, samozřejmě, byl nesmysl, protože Dospělý má atribut *jmeno* stejně jako ho má Člověk.

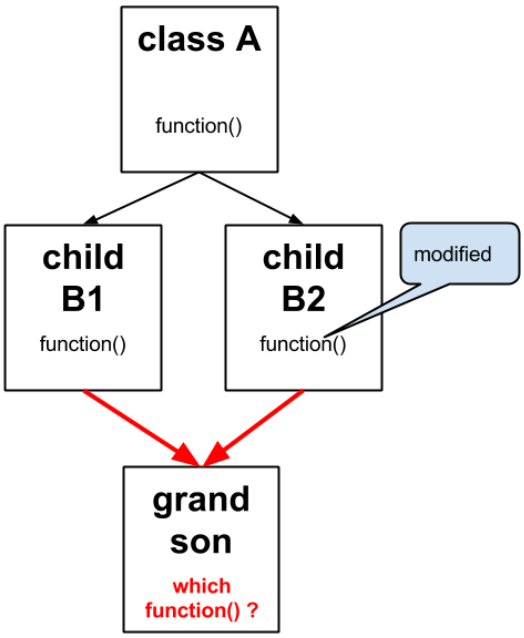
Proto je typová kontrola uvolněná tak, že všechno, co jde udělat s rodičem, jde udělat také s jeho potomkem.

Vysvětlení je prosté. Všechny atributy a všechny metody, které má rodič, jsou také obsaženy v potomkovi.

Ale pozor! Obráceně to neplatí. Potomek může obsahovat metody i atributy, které rodič nemá, a proto typová kompatibilita obráceným směrem neplatí!

Jen jeden rodič

Velký problém při dědění je, když vznikne tzv. diamantová struktura.



Když z rodiče A zdědíme 2 syny B1 a B2, je všechno v pořádku. V synovi B2 můžeme zmodifikovat funkci **func()** tak, aby dělala něco jiného. To je také v pořádku.

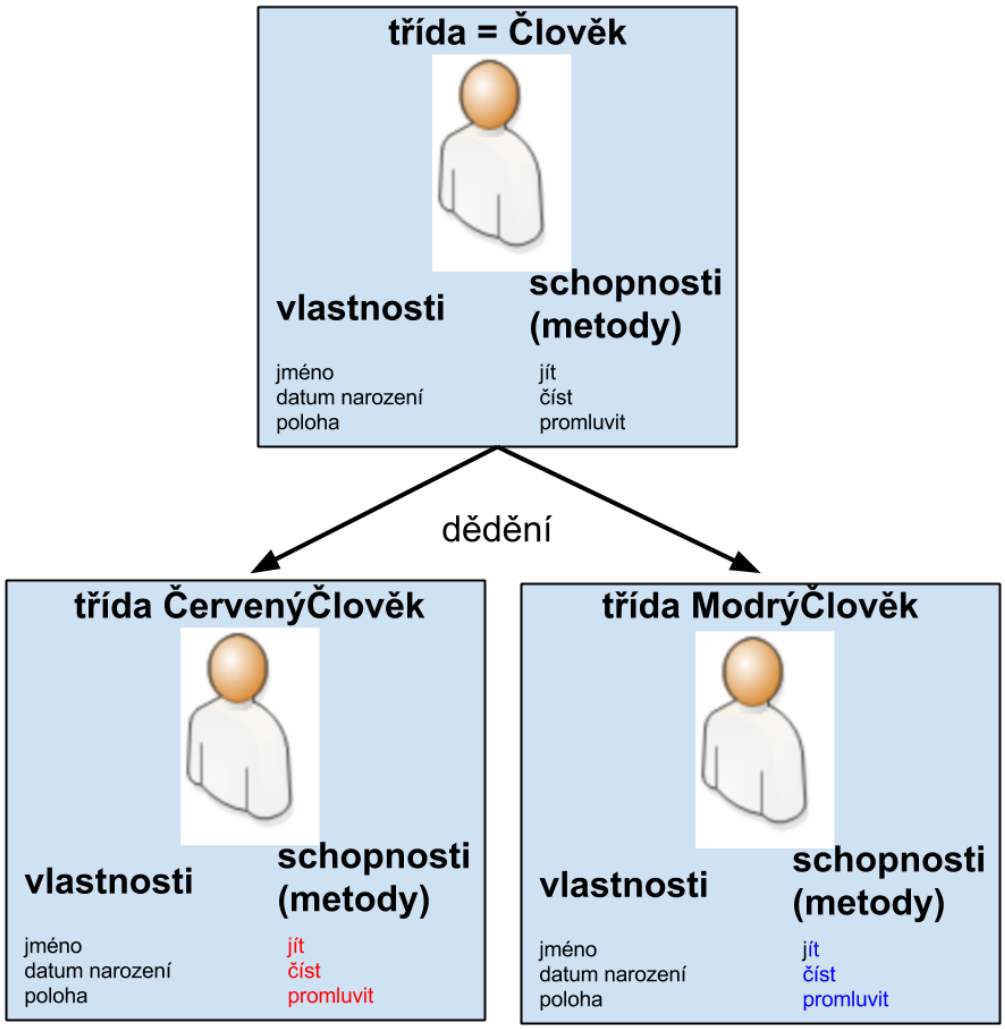
Ale kdyby vnuk měl dědit z B1 i z B2, byl by problém: která funkce u něj je ta správná? Ta původní, nebo ta modifikovaná?

Těmto potížím se obvykle předchází tak, že se zakáže dědění od více předků.

Polymorfismus

Polymorfismus velmi úzce souvisí s dědičností.

Ukázali jsme si, že při dědění můžeme změnit obsah, funkci metody. Představme si tedy, že ze třídy člověk zdědíme dvě nové třídy: ČervenýČlověk a ModrýČlověk. Budou se lišit tím, že jejich metody budou dělat něco úplně jiného, než metody z rodičovské třídy Člověk. To je perfektně legální.



Také jsme ukázali, že na místě, kam patří rodičovská třída, dá se použít libovolný jeji potomek. Na místě, kde jde použít `Člověk.číst()`, jde také použít `ČervenýČlověk.číst()` a také `ModrýČlověk.číst()`.

Takže do nějaké proměnné `x` můžeme vložit instanci jedné, druhé i třetí třídy. Všechny následující řádky jsou správně:

`x = instanceČlověk` nebo

`x = instanceČervenýČlověk` nebo

`x = instanceModrýČlověk`

Za běhu programu se, podle situace, do `x` může vložit jedna, nebo druhá, nebo třetí možnost. Říkáme, že proměnná se přiřazuje *dynamicky*.

Ale to je problém. Někde později v programu může být příkaz, například, `x.číst()`. Která ze tří metod *číst()* zde vlastně je zavolána? Volá se *x.číst()* nebo *x.číst()* anebo *x.číst()* ?

Volá se metoda té třídy, která zrovna v `x` je. To znamená, že volaná metoda se také mění dynamicky.

Předávání zpráv a události

Komunikace mezi objekty probíhá pomocí předávání zpráv, díky čemuž je syntaxe přehledná. Zpráva obvykle vypadá takto:

```
příjemce.jméno-metody(parametry).
```

Například **karel.pozdrav(sousedka)** by mohl způsobit, že instance **karel** pozdraví instanci **sousedka**.

Když se stane nějaká událost, třeba uživatel pohne myší, hardware vygeneruje přerušení a zpráva o tom se rozešle po celém počítači. Říkáme, že vznikla **událost**. Událostí každou vteřinu vzniká velmi mnoho.

Je na nás, zda na události chceme reagovat nebo nikoliv. Většinou se to dělá tak, že si vybereme jen události, které nás zajímají a na každou takovou událost napíšeme kousek programu, který ji zpracuje (obslouží). Těmto úsekům programu říkáme **handlery**.

Při programování pomocí IDE, třeba pro práci s grafikou, je zvykem, že tvůrce IDE dodá jakýsi polotovar, **framework**. Ten obsahuje všechnu základní funkcionalitu a všechny složité části. Na programátory aplikací pak zbývá, aby doplnili handlery, pomocí kterých se budou obsluhovat události.

Tento způsob programování se jmenuje **event-driven programming** čili **programování řízené událostmi**. My si ho vyzkoušíme při práci se *SceneBuilder*.

Cvičení

Vymyslete příklad, k čemu může být užitečný polymorfismus.

Ukažte, jak byste pomocí dědění realizovali OOP databázi knih.

Testy

Délka je pro třídu „vlak“ atribut-metoda-instance

Rychlost je pro třídu „vlak“ atribut-metoda-instance

Zastavit je pro třídu „vlak“ atribut-metoda-instance

Oranžový expres je pro třídu „vlak“ atribut-metoda-instance

Základní vlastnosti algoritmů jsou: >

Všeobecnost

Jednoznačnost

Konečnost

Rezultativnost

Správnost

Opakovatelnost

Efektivnost

Adaptivnost

Uzavřenost

Dědičnost

Přenositelnost

Operativnost

Senzitivnost

Máte třídu Auto a třídu Nákladák, která je zděděna z Auto. Zaškrtněte všechna pravdivá tvrzení.

S objektem třídy Naklaďák se dá nakládat, jako by byl třídy Auto.

S objektem třídy Auto se dá nakládat, jako by byl třídy Naklaďák.

S třídami Auto a Naklaďák se nikdy nemůže nakládat stejně, protože nejsou typově kompatibilní.

S třídami Auto a Naklaďák se vždycky může nakládat stejně, protože mají tentýž základ.

Zaškrtněte všechny správné zápisy (C a D jsou instance třídy uvedené v přednášce).

C.číst()

D.číst()

D.kouřit()

C.kouřit()

Třída může mít potomků nejvýše:

neomezený

1

2

3

žádný

Třída může mít rodičů nejvýše:

1

neomezený

2

3

žádný

Mějme rodičovskou třídu A a dceřinnou třídu B. Mějme proměnné *x* třídy A a *y* třídy B. Které příkazy jsou správně?

x = instanceA

x = instanceB

y = instanceB

y = instanceA

Obsah