Úvod do OOP.

Třída. Objekt. Zásady OOP. Abstrakce. Dědičnost. Kompozice. Polymorfismus. Objektový návrh programu.

Tomáš Bayer | bayertom@natur.cuni.cz

Katedra aplikované geoinformatiky a kartografie, Přírodovědecká fakulta UK.

Obsah přednášky

1. Objektově orientované programování 1/2

Paradigma OOP:

Metodický přístup zahrnující popis/řešení problému s využitím OOP.

V současné době jeden z nejčastěji používaných přístupů.

Existuje ve velké většině programovacích jazyků.

Charakteristika OOP:

- Vychází z objektového popisu řešeného problému (abstrakce skutečnosti).
- Poskytuje vhodné nástroje pro řešení tohoto problému.
- Není vázáno tak silně na algoritmus jako procedurální přístup.

Řešení problému s využitím OOP je **analogií** k „lidskému" řešení.

Postup v řadě případů efektivnější než procedurálním přístupu (ne vždy!).

Využívá konstrukce z procedurálního programování: funkce, proměnná.

Tyto skládány ve složitější celky.

2. Objektově orientované programování 2/2

Co umožňuje OOP programátorovi?

1. Abstrakci (zobecnění) pro modelování a řešení problémů.
2. Znovupoužitelný kód.
3. Kontrola přístupu k datům.
4. Minimalizace samovolných / nezamýšlených zásahů do kódu.
5. Udržení pořádku v identifikátorech.

OOP != třídy a objekty.

OOP představuje přístup, jak správně navrhnout strukturu programu (tj. jeho architekturu)

Program funkční a udržovatelný.

3. Objekt (Thing)

Objekty v reálném světě jsou hmotného charakteru.

Mají specifické vlastnosti ovlivňující jejich chování.

Objekty a OOP:

V OOP objekty abstrakcí svých hmotných protějšků.

Vycházejí z *objektového modelu* popisu problému.

Vykazují chování a mají určité vlastnosti.

V každém okamžiku lze popsat jejich stav.

Rozhraní:

Objekty spolu vzájemně komunikují přes *rozhraní*.

Prostřednictvím rozhraní objekt:

- přijímá požadavky od druhých objektů,
- zasílá požadavky druhým objektům,

a to ve formě zpráv (prováděné operace, výjimky).

Objekty se liší svým **vnitřním stavem**.

Mění v průběhu vykonávání programu.

4. Struktura objektu

Objekty jsou kombinací dat a funkcí, které nad těmito daty pracují.

Známy z procedurálního programování.

Objekt tvořen:

- Datovými položkami.
- Metodami.

Datové položky:

Ovlivňují vlastnosti objektu.

Představovány proměnnými různých datových typů.

Tato část objektu je soukromá (private)

Z vnějšku je před jinými objekty ukryta => *princip zapouzdření*.

Metody objektu:

Metody určují chování objektu.

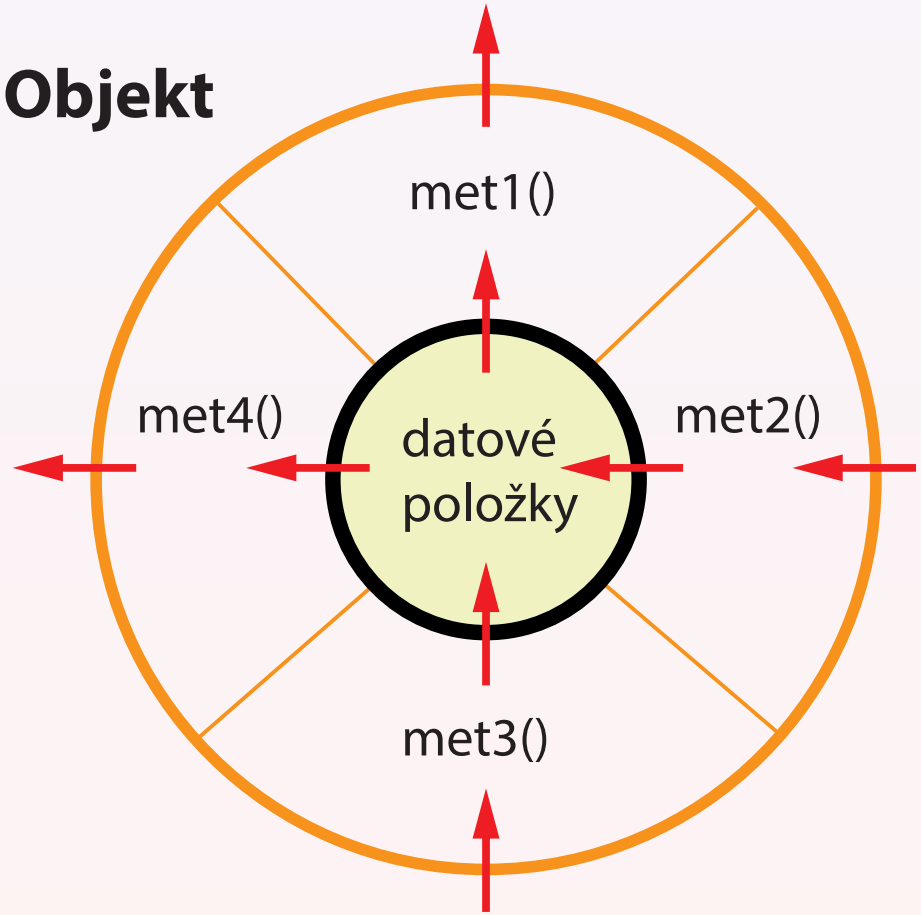
Definují operace, které je možno s daty provádět.

Tato část objektu je veřejná (public), tvoří s.

Poznámka: Metody mohou být deklarovány jako soukromé, nepředstavují část veřejného rozhraní.

5. Schematické znázornění objektu

Analogie s ovocem, jádro tvoří data, slupku metody.



6. Zapouzdření (Encapsulation)

Umožňuje zatajit vnitřní stav objektu před ostatními objekty.

Při operacích nemohou jiné objekty měnit stav objektu přímo $\Rightarrow$ rozhraní.

Nedochází k zanesení *nechtěných chyb* ovlivňujících funkčnost programu.

Metody objektu & zapouzdření:

Umožňují objektu komunikovat se svým okolím $\Rightarrow$ představují jeho *rozhraní*.

Datové položky objektu & zapouzdření:

Nejsou z vnějšku přímo viditelné.

Nemůžeme s nimi přímo manipulovat ani je vidět, k tomu lze využít pouze metody.

Výhody zapouzdření:

- *Zatajování informací*
  Datové položky nejsou přístupné jinému objektu z důvodu možného zavlečení chyby.
  Objekt má rozhraní umožňující komunikaci s ostatními objekty
  Chceme -li pracovat s objektem, musíme znát jeho rozhraní.
- *Modularita*
  Každý objekt lze udržovat a spravovat nezávisle na jiném objektu.
  Vylepšování funkcionality, aniž měníme rozhraní.

7. Zapouzdření a jeho realizace

Jak realizovat zapouzdření:

1. *Skrytí všech implementační detailů:*
   Veřejné třídy nepoužívají veřejné atributy.
   Opatrné používání `protected`: vede k porušení OOP.
2. *Nastavení přístupu k atributům:*
   Přístup s využitím pomocných metod, tzv. setterů a getterů (C++/Java).
   Alternativně s využitím dekorátorů (Python).
3. *Používání rozhraní pro vzájemnou komunikaci objektů.*
   Důsledek bodů 1, 2.
   Ne všechny metody automaticky součástí rozhraní.
   Chceme skrývat implementační detaily.
4. *Opatrné využívání dědičnosti:*
   Může porušit zapouzdření $\Rightarrow$ přístup `protected`.

Poznámka:

Zveřejňování rozhraní není časté, používáno ve speciálních případech.

Popření zásad OOP!

8. Komunikace objektů

Objekty spolu komunikují prostřednictvím zpráv.

K tomuto účelu používány metody tvořící *rozhraní*.

Objekt A vyžaduje po objektu B, aby vykonal nějakou činnost.

Zaslání zprávy v následujícím tvaru:

1. Objekt, na kterém se má akce provést.
2. Činnost, která bude vykonána pomocí metody.
3. Seznam parametrů předávaných metodě.
4. Seznam parametrů metodou vracených.
5. Typ návratové hodnoty.

Ve zprávě není informace jak to provést (pouze co provést).

Způsob implementace nemusí být znám.

Režie s vykonáním činnosti je dána objektu.

9. Třída (Class)

Třída je šablona sloužící k definici uživatelských datových typů.

Vychází z použitého objektového modelu.
Představuje abstrakci vlastností a funkcionality skutečných objektů.
Hovoříme o **abstraktních datových typech (ADT).**

Třída vs. objekt:
Třída určuje, jak bude objekt vypadat, a jak se bude chovat.
Představuje definici nového datového typu.

Objekty tvoří instance (proměnné) těchto datových typů.
Objekt vs. instance? Chápáno jako synonymum.

```
Třída vs. objekt = datový typ vs. proměnná
```

Třída tvořena komponentami $\Rightarrow$ **členy třídy**.
Představují je datové položky a metody. Třídy a jejich definice v Pythonu:

10. Třída, model v Pythonu

```
class Test:
    var1 = value                        #1. datova polozka tridy
    var2 = value                        #2. datova polozka tridy

    def __init__(self):                 #Implicitni (bezparametricky) inicializat
        self.var3 = value               #1. datova polozka instance
        self.var4 = value               #2. datova polozka instance
        do something

    def __init__(self, var3, var4):     #Explicitni (parametricky) inicializator
        self.var3 = value               #1. datova polozka instance
        self.var4 = value               #2. datova polozka instance
        do something

    def __del__(self):                  #Destruktor, vzdy bezparametricky
        do something                    #Nepovinny

    def f1(self, v1):                   #Metoda instance
        do something
    def f2(v1):                         #Metoda tridy (bez self)
        do something
```

Python nepřetěžuje inicializátory, výběr parametrického/neparametrického.

11. Datové položky třídy a instance

Existují dva typy datových položek.

- **Datové položky objektu**
  Z jedné třídy lze vytvořit libovolný počet instancí (objektů).
  Každý má vlastní sadu datových položek.

  Iniciovány při vytvoření objektu (instance) v konstruktoru/inicializátoru.
  Existují po celou dobu životnosti objektu.
  Zničeny v destruktoru.

  Nazývány jako *proměnné objektu/instance.*

  Proměnné jednoho objektu *nezávislé* na proměnných jiného objektu.

- **Datové položky třídy**
  Společné všem objektům (instancím) vytvořeným z jedné třídy.
  Nejsou vázány na konkrétní objekt.
  Společné všem instancím (objektům) třídy.

  Nazývány jako *statické datové položky.*

  Speciální funkce: sledování stavu/počtu objektů.

12. Konstruktor a inicializátor třídy

Při vytvoření objektu v Pythonu volán automaticky konstruktor.

Po vytvoření objektu volán automaticky inicializátor.

```
__new__(self):    #Konstruktor: vytvoreni noveho objektu
__init__(self):   #Inicializator: Inicializace datovych polozek
```

Inicializátor třídy:

Z vnějšku k datovým položkám nelze přistupovat $\Rightarrow$ inicializace v inicializátoru.

Inicializátor nelze přetěžovat.

První parametr `self`: odkaz na volající objekt.

- *Bezparametrický inicializátor*
  Nepředáváme mu žádné o informace datových položkách instance.
  Všechny instance jím vytvořené tímto způsobem mají stejné vlastnosti.
  ```
  def __init__(self):
  ```
- *Parametrický inicializátor*
  Z vnějšku předány informace o hodnotách datových položek instance.
  Každá jím vytvořená instance může mít jiné vlastnosti.
  ```
  def __init__(self, var1,..., varn):
  ```

Destruktor třídy: __del__(self)

Speciální metoda volaná automaticky při mazání objektu.

13. Metody třídy a instance

Existují datové položky objektu a třídy.

Metody objektu (instance):

Voláme vždy pro konkrétní instanci (objekt):

```
def function(self, param):   #Definice, první parametr self
object.function(param)       #Teckova notace, volani pro objekt
```

Pracuje s proměnnými objektu i třídy.

Představuje zprávu poslanou konkrétní instanci.

Lze je volat v případě, kdy je vytvořena konkrétní objekt.

Metody třídy:

Nejsou volány pro konkrétní objekt:

```
def function(param):         #Definice, nepouziva self
class_name.funtion(param)    #Teckova notace, volani pro tridu
```

Představují zprávu zaslanou třídě jako celku.

Pracují pouze s proměnnými třídy, nikoliv s proměnnými instance.

Nazývány *statickými metodami.*

Používány ve speciálních případech, sledování stavu instancí.

14. Návrh třídy Point I (chybně)

Bezparametrický inicializátor:

```
class Point:
    id = 0;                 #Datova polozka tridy

    def __init__(self): #Inicializator
        Point.id = Point.id + 1
        self.x = 0
        self.y = 0

    def print(self):    #Tisk
        print("id = " +str(Point.id) + ",
              x=" + str(self.x) + ", y=" + str(self.y))
p1 = Point()
p1.print()
p2 = Point()
p2.print()
p1.print()
```

Výsledek:

```
>>> id = 1, x=0, y=0
>>> id = 2, x=0, y=0
>>> id = 2, x=0, y=0  #Oba body maji stejne cislo
```

15. Návrh třídy Point II (OK?)

Bezparametrický inicializátor:

```
class Point:
    counter = 0;                                #Datova polozka tridy

    def __init__(self):                         #Inicializator
        self.id = Point.counter                 #Aktualni hodnota pocitadla
        self.x = 0
        self.y = 0
        Point.counter = Point.counter + 1       #Inkrementace pocitadla

    def print(self):                            #Tisk
        print("id = " +str(self.id) + ",
              x=" + str(self.x) + ", y=" + str(self.y))
p1 = Point()
p1.print()
p2 = Point()
p2.print()
p1.print()
```

Výsledek:

```
>>> id = 0, x=0, y=0
>>> id = 1, x=0, y=0
```

16. Návrh třídy Point III (OK?)

Univerzální varianta s parametrickým inicializátorem:

```
class Point:
    counter = 0;                              #Datova polozka tridy

    def __init__(self, x = 0, y = 0):         #Inicializator
        self.id = Point.counter               #Aktualni hodnota pocitadla
        self.x = x
        self.y = y
        Point.counter = Point.counter + 1

    def print(self):                          #Tisk
        print("id = " +str(self.id) + ",
              x=" + str(self.x) + ", y=" + str(self.y))
p1 = Point(10, 10)
p1.print()
p2 = Point(30, 70)
p2.print()
```

Výsledek:

```
>>> id = 1, x=10, y=10
>>> id = 2, x=30, y=70
```

17. Přístup ke členům třídy

V praxi využíván princip zapouzdření.
Některé členy třídy označeny jako soukromé (*private*), jiné jako veřejné (*public*).

Modifikátory přístupu:
Řídí přístup ke členům třídy (datové položky + metody).
Nastavení, ke kterým částem objektu bude možno přistupovat z vnitřku/vnějšku.
Určují, které části objektu lze modifikovat a jak.

Modifikátor *public*: přístup z libovolné třídy.
Modifikátor *private*: přístup pouze z třídy, kde byly deklarovány.
Modifikátor *protected*: přístup ze třídy, byly deklarovány + z odvozených tříd.

Soukromé členy třídy deklarovány zpravidla jako *private*.
Veřejné členy třídy jako *public* .
U dědičnosti *protected*.

```
a       #Polozka je public
_a      #Polozka je protected
__a     #Polozka je private
```

18. Realizace principu zapouzdření (1/2)

Datové položky nedodržují princip zapouzdření:

```
p1 = Point(10, 10)
p1.x = 7    #Pristup z vnejsku, poruseni principu
```

Nutnost nastavení privátního přístupu: znak __

```
class Point:
    __counter = 0;                          #Nyni privatni

    def __init__(self, x = 0, y = 0):
        self.__id = Point.__counter         #Nyni privatni
        self.__x = x                        #Nyni privatni
        self.__y = y                        #Nyni privatni
       ...
```

Nyní nelze:

```
p1.x = 7    #Promenna x je nyni privatni
```

Jak získat/změnit hodnoty souřadnic?

- Vytvoření veřejných metod (gettery, settery)
- Použití dekorátorů.

19. Použití veřejných metod

Použití pomocných metod pro zjištění hodnot datových položek a jejich změnu.

Getter:

Zjištění hodnot datových položek.

Nemodifikuje datovou položku.

```
def getX(self):              #Verejna metoda
    return self.__x
....
x = p1.getX()                #Pouziti getteru, dotaz na hodnotu
```

Setter:

Změna hodnot datových položek.

Nemá návratovou hodnotu.

```
def setX(self, x):           #Verejna metoda
    self.__x = x
...
p1.setX(37)                  #Pouziti setteru, nastaveni hodnoty
```

Zpravidla každá datová položka disponuje vlastním setterem/getterem.

Tento přístup běžný v C++/Javě, Python preferuje `property`.

21. Použití dekorátorů

= příznak @

Specifická změna syntaxe umožňující ovlivňovat chování funkcí, metod, tříd.

```
@decorator
def function():
    some_code
```

Využití dekorátoru `property`, alternativa ke getteru:

```
@property                   #Pouziti dekoratoru @property
def x(self):
    return self.__x
...
x = p1.x
```

Využití dekorátoru `var.setter`, alternativa k setteru:

```
@x.setter                   #Pouziti dekoratoru @x.setter
def x(self, x):
    self.__x = x
...
p1.x = 37
```

Jednodušší syntaxe než gettery/settery, preferováno v Pythonu.

```
p1.id = 12                  #Nelze pouzit, nedefinovan dekorator
```

22. Objekty jako parametry funkcí

Objekty můžeme předávat jako skutečné parametry funkcí:

```
def dist(self, p1, p2):            #Vzdalenost mezi 2 body
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    return sqrt(dx * dx + dy * dy)
....
d = dist(p1, p2)
```

Vracení objektů vytvořených funkcemi, v1

```
def mid(self, p1, p2, pmid):
    pmid.x = (p1.x + p2.x)/2
    pmid.y = (p1.y + p2.y)/2
....
mid(p1, p2, pm)
```

Vracení objektů vytvořených funkcemi, v2

```
def dist(self, p1, p2):
    pmid = Point((p1.x + p2.x)/2, (p1.y + p2.y)/2)
    return pmid                    #Vracen novy bod
....
pm = mid(p1, p2)
```

23. Statická funkce

Statické funkce možno volat bez vytvořeného objektu.

Využití tečkové notace, místo objektu název třídy

```
class_name.method()
```

Mohou pracovat pouze se statickými nebo lokálními proměnnými.

Využíván dekorátor `@staticmethod`.

```
class Algorithms:
    @staticmethod                #Pouziti dekoratoru @staticmethod
    def mid2(p1, p2):            #Prvni parametr neni self
        pmid = Point((p1.x + p2.x)/2, (p1.y + p2.y)/2)
        return pmid
```

Ukázka volání funkce:

```
p1 = Point(10, 10)
p2 = Point(30, 70)
pm = Algorithms.mid2(p1, p2)  #Trida.metoda
```

24. Zásady OOP:

Základní zásady OOP:

- *Zapouzdření (Encapsulation)*
  Zatajení vnitřní stavu objektu.
- *Abstrakce (Abstraction).*
  Zobecnění, oproštění se od nedůležitých detailů.
- *Kompozice (Composition)*
  Opětovné použití implementace.
- *Dědičnost (Inheritance)*
  Opětovné použití rozhraní.
- *Polymorfismus (Polymorphism)*
  Objekty reagují na stejný podnět dle svého typu.

25. Abstrakce a programování

Projevuje se ve 3 hlavních směrech:

1. *Konceptuální zjednodušení*
   Konceptuální zjednodušení problému.
   Vlastnosti a funkce skutečného objektu přenášeny na nehmotný objekt.
   Zanedbání méně podstatných či nepodstatných detailů, vlastností.
   Spojeno s objektovým modelem.

2. *Abstrahování se od detailů*
   Abstrakce od detailů, jakým způsobem pracují jednotlivé objekty „uvnitř”.
   Pro práci s objektem stačí znalost rozhraní, kterým objekt komunikuje.
   Programátor objektu sděluje, co má udělat, a nikoliv jak to má udělat.
   Spojeno se zapouzdřením (Encapsulation).

3. *Zobecnění*
   Hledání společných vlastností, zastřešujících nadtříd.
   Základním rys objektového návrhu programu.
   Spojeno s dědičností (Inheritance).

26. Třída & abstrakce

Třída je nositelem abstrakce.

Abstrahuje programátora od implementačních detailů, které ukrývá.

Rozhraní důležitou částí návrhu třídy.

1. *Třída není jen datovým kontejnerem*
   Zahrnuje funkcionalitu nad daty.
2. *Třída má vykonávat pouze jednu věc*
   A to dobře :-)
   Nenavrhovat univerzální třídy!
3. *Nepřidávat do třídy další metody přímo nesouvisející s třídou*
   Eroze návrhu, rozmělnění.
4. *Dodržovat jednotnou úroveň abstrakce komponent*
   Nekombinovat obecné u úzce specializované třídy

Rozčlenění problému do tříd, výhody:

- Funkcionalita programu je zjevnější.
- Operace mají samovysvětlující charakter.

27. Kompozice

Složité objekty chápány jako kolekce objektů jednodušších.
Akce nad nimi mohou být redukovány na jednodušší.
Postup se opakuje tak dlouho, dokud objekty/akce nejsou triviální.

Kompozice představuje *opakované použití implementace*.
Kompozice je jedna ze základních vlastností OOP.
Výsledkem je znovupoužitelný kód.
Kompozice (skládání) = použití instance jedné třídy v definici druhé třídy.
Třídu lze "skládat" z více objektů jiných tříd.

Kompozice umožňuje použití hierarchického přístupu při návrhu tříd.
Přístup, při kterém z jednodušších komponent skládáme složitější komponenty.

Agregace: agregované části objektu mohou existovat i bez objektu.
Kompozice: kompozity nemohou existovat bez objektu.

28. Ukázka kompozice, třída Line

Třída Line:

```
class Line:
    def __init__(self, p1, p2):    #Pocatecni a koncovy bod
        self.__p1 = p1
        self.__p2 = p2

    def Print(self):
        self.__p1.Print()
        self.__p2.Print()
```

Volání funkce:

```
line = Line(p1, p2)
line.Print()
```

29. Ukázka kompozice, třída Polygon

Třída Polygon: body reprezentovány *n*-ticí.

Využití pozičních argumentů *.

Formální parametry automaticky konvertovány na *n*-tici.

```
class Polygon:
    def __init__(self, *points):     #Pozicni argument,
        self.__points = points        #vytvori n-tici

    def print(self):
        for p in self.__points:
            p.Print()
```

Volání funkce:

```
pol = Polygon(p1, p2, p3, p4)         #Vytvoreni n-tice
pol.Print()                           #points = (p1, p2, p3, p4)
```

30. Abstrakce a dědičnost

Zobecnění:

Nalezení společného, nadřazeného, zastřešujícího významu/pojmu pro skupinu prvků.

Postup od konkrétního k obecnému.

Příklady zobecnění:

- Mrkev, kedlubna, květák, cibule, pórek $\Rightarrow$ zelenina.
- Zelenina, ovoce, obilniny $\Rightarrow$ potraviny.

Existuje několik úrovní zobecnění lišící se mírou podrobnosti zobecnění.

Abstrakce:

Při abstrakci zamlčujeme informace/vlastnosti, které je obtížné zobecnit.

Oproštění se od detailů, které nemusejí být podstatné.

Příklad abstrakce*:*

Švestka, třešeň, jablko, kokosový ořech, ananas $\Rightarrow$ ovoce.

Jako přílohu k jídlu si vezmu ovoce (jablko, avšak i kysané zelí), abstrakce od chuti.

Zanedbávány mají být pouze méně důležité vlastnosti $\Rightarrow$ tj. abstrahujeme se od nich.

31. Dědičnost

Modeluje vztah generalizace/specializace mezi dvěma objekty, O1 a O2:

- Objekt O1 je zobecněním objektu O2 (tj. jeho generalizací).
- Objekt O2 je zvláštním případem objektu O1 (tj. jeho specializací).

Specializované objekty přebírají vlastnosti/chování od objektů, ze kterých jsou odvozeny.

Dědičnost představuje *opětovné použití rozhraní*.

Umožňuje používat při návrhu tříd hierarchický přístup.

Princip dědičnosti:

Třídy na < úrovni mohou dědit vlastnosti a chování tříd na > úrovni.

Novou třídu lze odvodit z existující třídy přidáním nové funkcionality a vlastností.

V užším slova smyslu pro dědičnost + polymorfismus používán termín *objektově orientované programování*.

Původní třída = rodičovská, bázová třída.

Odvozená třída = třída potomků.

Jednoduchá vs. vícenásobná dědičnost.

Jednoduchá dědičnost: odvozená třída dědí pouze z jedné rodičovské třídy.

Vícenásobné dědičnost: odvozená třída potomkem více rodičovských tříd.

Ne všechny programovací jazyky ji podporují (Python ano).

32. Předek, potomek

Předek a potomek:

Děděním vzniká z předka potomek.

Potomek zdědí od předka všechny vlastnosti.

Zpravidla definujeme vlastnosti a metody, které jsou pro něj specifické.

Potomek tedy umí to samé, co předek, plus něco navíc.

Potomek může vždy zastoupit předka.

Dědická hierarchie:

Posloupnost rodičů a jejich potomků vytváří dědickou hierarchii.

Lze ji znázornit stromovou strukturou.

Předefinování metody:

Metoda rodičovské třídy může být v odvozené třídě předefinována.

Je vybavena novou funkcionalitou.

Předefinovaná metoda má stejnou hlavičku, liší se pouze “výkonnou” částí.

33. Dědičnost, výhody, použití

Dědit lze jak rozhraní, tak implementaci (kompozice).

Při dědění *rozhraní* není děděn kód, ale pouze signatura rozhraní:

- prototypy metod,
- datové položky.

V odvozené třídě definována nová funkcionalita: *předefinování metody*.

Při dědění *implementace* děděn kód včetně signatur.

Výhody dědičnosti:

- *Omezování duplicitního kódu*
  Společné rysy rodičovská i odvozená třída sdílejí.
  Specifické vlastnosti definovány v odvozené třídě.
- *Společné zacházení s více třídami*
  Společné zacházení s třídami pro operace, ve kterých vykazují společnou funkcionalitu.

Dědičnost lze zakázat, finální třídy (Python podporuje).

34. Dědičnost v Pythonu

Jednoduchá dědičnost: třída potomkem jedné třídy

```
class class_name(base_class):
```

Vícenásobná dědičnost: třída potomkem více tříd

```
class class_name(base_class1,..., base_classn):
```

Hierarchie volání inicializátorů:

1. *V inicializátoru odvozené třídy volán inicializátor rodičovské třídy*

   Zajišťuje inicializaci datových položek rodičovské třídy.

   ```
   super().__init__(param1,..., paramn)
   ```

   Definice datových položek odvozené třídy může záviset na datových položkách rodičovské třídy !!!.

2. *Inicializace datových položky třídy odvozené*
   Následně inicializovány datové položky odvozené třídy.

Metoda `super()`:

Umožňuje volat metody rodičovské třídy, tj. i inicializátory.

Datové položky v rodičovské/odvozené třídě:

Často používán atribut `protected`.

V odvozených třídách se chovají jako `public`.

35. Ukázka rodičovské třídy

Třída GO reprezentující obecný grafický objekt:

```
class GO:
    def __init__(self, color = 0, width = 0, layer = 0):
        self._color = color             #Protected
        self._width = width             #Protected
        self._layer = layer             #Protected

    @property
    def color(self):
        return self._color

    @color.setter
    def color(self, color):
        self._color = color

    def print(self):
        print("col = " + str(self._color) + ", width=" +
        str(self._width) + ", layer=" + str(self._layer))
```

36. Ukázka odvozené třídy

Třídy `Point` potomkem třídy `GO`.

Konstruktoru odvozené třídy předáváme i parametry rodičovské třídy.

Tyto používá při inicializaci datových položek rodičovské třídy.

```
class Point(GO):
    __counter = 0;

    def __init__(self, color = 0, width = 0,
                 layer = 0, x = 0, y = 0):
        super().__init__(self, color, layer,     #1. Inicializator rodicovske
                         width)                  #tridy
        self.__id = Point.__counter              #2. Inicializator odvozene
        self.__x = 0                             #tridy
        self.__y = 0
        Point.__counter = Point.__counter + 1
```

Vytvoření nového objektu:

Předáváme parametry pro inicializaci položek rodičovské i odvozené třídy.

```
p = Point(0, 1, 1, 25, -37)   #Inicializace polozek rodicovske/odvozene tridy
```

37. Abstraktní třídy a metody

Abstraktní třída:

Nelze od ní vytvářet instance.

Může obsahovat proměnné, metody, abstraktní metody.

V Pythonu potomek třídy ABC.

```
from abc import ABC, abstractmethod
    class class_name(ABC):
```

Použití při tvorbě nadtříd:

- zastřešují společné vlastnosti,
- fakticky s nimi nepracujeme (nepotřebujeme jejich instance).

Abstraktní metoda:

Vyskytuje se pouze v abstraktních třídách.

Musí být v odvozené třídě předefinována.

Nemají žádný výkonný kód.

Použití dekorátoru `@abstractmethod`.

```
@abstractmethod
def f(self, var1, varn):     #Musi byt predefinovana
        pass                 #Zadny vykonny kod
```

38. Ukázka abstraktní třídy

Abstraktní třída pro reprezentaci grafických objektů:

```
from abc import ABC, abstractmethod    #Import modulu
class AGO(ABC):                        #Abstract base class
    def __init__(self, color = 0, width = 0, layer = 0):
        self._color = color        #Protected
        self._width = width        #Protected
        self._layer = layer        #Protected

    @abstractmethod                #Pouziti dekoratoru
    def print(self):
        pass
```

Snaha o vytvoření instance této třídy:

```
ag = AGO()
>>> TypeError: Can't instantiate abstract class AGO
>>> with abstract methods print
```

39. Odvozená třída

Třída odvozená od abstraktní třídy předefinovává všechny abstraktní metody.

```
class Point(AGO):
    counter = 0;

    def __init__(self, color = 0, width = 0, layer = 0, x = 0, y = 0):
        super().__init__(self, color, witdh, layer)  #Inicializator rodicovske

        self.__id = Point.counter
        self.__x = x
        self.__y = y
        Point.counter = Point.counter + 1

    def print(self):              #Predefinovana abstraktni metoda
        super.print()             #Metoda rodicovske tridy
        print("id = " + str(self.__id) + ", x=" + str(self.__x) +
              ", y=" + str(self.__y))
```

Vytvoření objektu:

```
p = Point3(1, 2, 3, 35, 35)
p.Print()
>>> col = 1, width=2, layer=3
>>> id = 4, x=35, y=35
```

40. Dědičnost vs. kompozice I.

ISA-HAS test nutná avšak nikoliv postačující.

Použití dědičnosti:

Nová třída je specializací původní třídy.

Použijeme v případě, kdy mezi původní a odvozenou třídou existuje vztah “JE”.

“Je osobní auto také auto?” Odpověď zní: ano.

Dotaz zda má osobní auto auto postrádá smysl.

Použití kompozice (agregace):

Nová třída má v sobě komponenty původní třídy.

Použijeme v případě, pokud mezi původní a odvozenou třídou existuje vztah “MÁ”.

Třída `Point` obsahuje dvě datové položky x,y představující souřadnice bodu.

Třída `Line` bude obsahuje dvě datové položky typu `Point`.

Ptáme se: “Má linie body?” Odpověď je ano.

Dotaz, zda je linie bod, postrádá smysl.

41. Liskov Substitutional Princip

Barbara Liskov (1994), pravidla používání dědičnosti.

Předpoklady: platí před vykonáním nějaké metody objektu (včetně inicializátoru).

Následné podmínky: platí po vykonání nějaké metody objektu.

Invarianty: platí po celou dobu existence objektu

1. Předpoklady nesmí být zesilovány v podtřídách (Elipsa -> Kruh).
2. Následné podmínky nesmí být oslabovány v podtřídách (Kruh -> Elipsa).
3. Invarianty rodičovské třídy zachovány i v odvozených třídách.

Kovariance: Použití konkrétnějšího (odvozeného) typu.

Kontravariance: Použití abstraktnějšího (nadřazeného) typu.

1. Kovariance návratových typů v odvozených třídách: návratový parametr rodičovské třídy zastoupen libovolnou odvozenou třídou.
2. Kontravariance argumentů metod v odvozených třídách: předávaný parametr odvozené třídy zastoupen parametrem rodičovské třídy.
3. Podtřídy nemohou vyvolat řádné jiné výjimky než ty, použité v rodičovské třídě, nebo třídy odvozené od těchto výjimek.

Důsledky:

1. Dědičnost by měla být použita pouze v případě, že odvozená třída je specializací rodičovské třídy
2. Existuje vztah is.
3. Všude, kde lze použít třída, musí být použitelný i její potomek tak, aby uživatel nepoznal rozdíl (polymorfismus).
4. Vztah is musí být trvalý.

42. Ukázka chybného návrhu

Rodičovská třída:

```
class Ellipse:
    def __init__(self, a, b): #OK
        self.__a = a
        self.__b = b
    def resize(self, a, b):
        self.__a = a
        self.__b = b
```

Odvozená třída:

```
class Circle(Ellipse): #OK
    def __init__(self, r):
        super().__init__(self, r, r)
    def resize(self, a, b):
        if a != b:
            print('Error, a != b')  #Preconditions cannot be strengthene
        else:
            super().__init__(self, a, a)
```

43. Dědičnost / kompozice a návrh tříd

Při návrhu nutno zohlednit:

1. Volba viditelnosti datových položek třídy.
2. Volba viditelnosti metod.
3. Volba abstraktních metod.
4. Volba virtuálních metod.
5. Volba abstraktních tříd.
6. Volba finálních tříd.

Zásady:

- Společné vlastnosti tříd umisťovat co nejvýše v hierarchii.
  Tvoří společné rozhraní.
- Nepoužívat příliš složité hierarchie.
  Max. 4 úrovně, jinak vede k nepřehlednosti.
- Zamyslet se nad použitím třídy s jedním potomkem.
  Má taková třída smysl?

44. Dědičnost / kompozice a problémy

Možné problémy a úskalí kompozice / dědičnosti:

- Nadužívání kompozice i dědičnosti v případech, kdy to není potřeba.
  Příliš složitý návrh modelu.
- Pozor na rigidní výklady IS-HAS
  Např. vztah `Point2D` a `Point3D`, `Circle` a `Ellipse` - ani dědičnost, ani kompozice!
- Pokud to je možné, dávat přednost kompozici před dědičností.
- Omezit používání vícenásobné dědičnosti.
  Dle některých názorů by neměla být vůbec používána.
- Při používání dědičnosti dochází k porušení zapouzdření.
  Týká se použití `protected` u datových položek.

45. Polymorfismus

Objekty v dědické hierarchii reagují na stejný podnět různě, a to v závislosti na typu.

Každá operace může mít více implementací.

Konkrétní implementace zvolena v závislosti na typu objektu, s nímž se operace provádí.

Využití polymorfizmu:

- *Přetěžování funkcí*
  Stejný identifikátor, liší se počtem / typem parametrů (Python nepodporuje).
- *Přetěžování metod*
  Různá implementace operace v odvozených třídách.
- *Dědičnost rozhraní*
  Rodičovská třída použita k definici rozhraní (společné vlastnosti).
  Odvozené třídy implementují odlišnosti.
- *Generalizace*
  Schopnost společného zacházení s objekty prostřednictvím nadtřídy.

Pro implementaci polymorfismu je používána pozdní (dynamická) vazba.

46. Polymorfismus, statická a dynamická vazba.

Důsledek polymorfismu:

Instance různých tříd v dědické hierarchii na stejný podnět reagují různě.

V místě, kde je očekávána instance nějaké třídy, lze použít instanci libovolné odvozené třídy.

Potomek může vždy zastoupit předka.

Tato instance se může chovat jinak, než by se chovala instance rodičovské třídy.

Platí pouze v rozsahu daném popisem rozhraním.

Časná (statická) vazba *(Early Binding):*

O použití metody předka/potomka rozhodnuto při překladu.

Nezohlední se „typ” objektu.

Pozdní (dynamická) vazba *(Late Binding):*

O použití metody předka/potomka rozhodnuto až v době běhu programu.

Zohledňuje „typ” objektu.

Java/Python používají pozdní vazbu, v C++ nutné klíčové slovo `virtual`.

47. Odvozená třída Line

Ve třídě `GO` definována metoda `print()`.

Od třídy `GO` odvozena třída `Line`.
V ní předefinována metoda `print()`.

Pro objekt třídy `Line` volána „její” metoda `print()`

```
class Line (GO):
    def __init__(self, color, width, layer, x1, y1, x2, y2):
        super().__init__(color, width, layer)
        self.__p1 = Point(x1, y1)
        self.__p2 = Point(x2, y2)

    def print(self):                    #Overriden method
        self.__p1.Print()
        self.__p2.Print()
```

V těle volána metoda `print()` třídy `Point`.

48. Ukázka polymorfismu

Dle typu objektu volána metoda:

```
g = GO(1, 2, 3)
g.Print()          #Metoda rodicovske tridy GO
p = Point(1, 2, 3, 35, 35)
p.Print()          #Metoda odvozene tridy Point
l = Line2(1, 2, 3, 0, 0, 10, 10)
l.print()          #Metoda odvozene tridy Line
```

Ukázka generalizace: společné zacházení s objekty

```
gos = [g, p, l]
for go in gos:
    go.grint()
```

V kontejneru současně objekty rodičovské i odvozené třídy.

Podporuje většina programovacích jazyků.

Python: dynamicky typovaný, v kontejneru různé typy.

49. Hluboká a mělká kopie

Objekty v Pythonu chápány jako mutable (mohou měnit stav).
Při přiřazení není automaticky vytvářen nový objekt (zachování ID).

Mělká kopie objektu (Shallow Copy):
U atributu objektového typu se kopíruje pouze odkaz na tento objekt.
Nejsou prováděny žádné nové alokace paměti.
Kopie je závislá na originálu.

Hluboká kopie objektu (Deep Copy):
Provádí fyzické překopírování všech atributů, nikoliv pouze odkazů.
Je-li atribut objektového typu, je vytvořena kopie tohoto objektu.
Kopie je nezávislá na originálu (kopírování hodnot).

Python 3 podporuje obě varianty kopírování.
Nutno importovat modul `copy`.

50. Ukázka: Shallow/Deep Copy

Mělké kopírování:

```
p1 = Point(10, 10)
p2 = p1                          #Shallow copy of p1
id(p1)                           #Get ID of p1
>>> 2898181520640
id(p2)                           #Get ID of p2
>>> 2898181520640
p1.x = 5
p1.print(); p2.print();
>>> id = 0, x = 5, y = 10
>>> id = 0, x = 5, y = 10        #Unique ID and coordinates
```

Hluboké kopírování: metoda `deepcopy()`

```
import copy
from copy import deepcopy

p1 = Point(10, 10)
p2 = deepcopy(p1)                #Deep copy of p1
id(p2)                           #Get ID of p2
>>> 2289816783504
p1.x = 5
p1.print(); p2.print();
>>> id = 0, x = 5, y = 10
>>> id = 1, x = 10, y = 10       #Different ID and coordinates
```

51. Porovnání dvou instancí

U základních datových typů definovány relační operace:

```
-709 < -3.7 < 1.5 == 1.5 < 59.8 < 100.3
„N” == „N” < „O” < „P” < „T” < „Y”
```

Třída představuje definici vlastního uživatelského typu.

Jak porovnat dva uživatelské datové typy?

Vytvoříme -li body:

```
p1 = Point(0,0)
p2 = Point(10,10)
p3 = Point(0,5)
p4 = Point(10,10)
p5 = Point(-3,-7)
```

Který z bodů je „větší”/”menší”!?

Kdy jsou dva body rovny?

Nelze určit, nutné stanovit kritérium.

Definice operátorů:

```
==    __eq__ (self, other)    equal
<     __lt__ (self, other)    less than
>     __gt__ (self, other)    greater than
```

52. Definice operátorů <, >,==

Uvnitř třídy nutno předefinovat operátory <, >, ==.

V každé třídě pouze jedna jeho definice, nelze přetížit.

Operátor <

```
def __lt__(self, other):
   return self.__x < other.__x    #Porovnani souradnic x1, x2

def __lt__(self, other):
   return (self.__x < other.__x) or (self.__x == other.__x)
        and (self.__y < other.__y)  #Pokud x1 == x2, porovnej y1 ? y2
```

Operátor ==

```
def __eq__(self, other):
    return (self.__x == other.__x) and (self.__y == other.__y)
```

Ukázka:

```
print (p1 < p2)
>>> True
print (p1 < p3)
>>> False
print (p2 == p4)
>>> True
```

53. Třídění objektů v kolekcích

Objekty v dynamických datových strukturách lze třídit vzestupně/sestupně:

```
collection.sort(key, reverse=True/False)            #In place (mutable)
res = sorted(collection, key, reverse=True/False)   #New collection (immutabl
```

Objekty nějaké třídy je možné třídit:

1. *Pokud jsou v třídě definovány definovány operátory <, ==*
   Není nutné specifikovat datovou položku, podle které třídíme.
   Nevýhoda: objekty lze třídit pouze podle 1 kritéria

   ```
   points.sort()                          #No key defined
   points.sort(reverse = True)            #Descendent sort
   ```

2. *Pokud je specifikován atribut*
   Atribut reprezentuje klíč `'key'`, jednoduchý typ.
   Jsou pro něj implicitně definovány operátory

   ```
   points.sort(key=lambda k: k.y)         #Using lambda exressions
   from operator import attrgetter
   points.sort(key= attrgetter('y'))      #Using the property
   ```

3. *Pokud je definován komparátor*
   Obecná funkce, umožňuje definovat libovolné (složitější) kritérium.
   Objekty lze třídit podle „libovolného” komparátoru více způsoby.

Sortable collections:

List (sort, sorted), Tuple (immutable, sorted), Dictionary (sorted, jen dle klíčů)

54. Vlastnosti komparační funkce

Komparační funkce $C(x, y)$ rozlišuje 3 stavy:

$$C(x,y) = \begin{cases} -1, & x < y, \\ 0, & x = y, \\ 1, & x > y. \end{cases}$$

Požadavky:

1. *Reflexivita*

   $$C(x,x) = 0, \qquad \forall x.$$

   Příklad: $x = x$.

2. *Antisymetrie*

   $$C(x,y) = -C(y,x), \qquad \forall x, y.$$

   Příklad: Pokud $x < y$, $\Rightarrow y > x$.

3. *Tranzitivita*

   $$C(x,y) = C(y,z) = a \Rightarrow C(x,z) = a, \qquad \forall x, y, z.$$

   Příklad: Pokud $x < y$, a $y < z \Rightarrow x < z$.

Vyžadovány 1), 3).

Např. 2) nelze splnit pro metriky.

55. Vytvoření komparační funkce (1/2)

Vrací hodnotu kritéria, které bude použito jako klíč.

Definována vně třídy, jejíž objekty porovnáváme.

Příklady:

1. Setřídění množiny bodů P dle vzdálenosti od $p = [x, y]$.
2. Setřídění množiny bodů P dle vzdálenosti od těžiště T.

Vstupní body $P = \{p_i\}_{i=1}^{n}$, $p_i = [x_i, y_i]$, těžiště $T = [x_T, y_T]$, kde

$$x_T = \frac{1}{n}\sum_{i=1}^{n} x_i, \qquad y_T = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

Vzdálenost

$$\left[\|p_i, p_j\|_2 = (x_i - x_j)^2 + (y_i - y_j)^2\right]^{1/2}.$$

Reflexivita: $\|p_i, p_i\|_2 = 0$.

Symetrie: $\|p_i, p_j\|_2 = \|p_j, p_i\|_2$.

Tranzitivita: $\|p_i, p_j\|_2 + \|p_j, p_k\|_2 \leq \|p_i, p_k\|_2$ ($\triangle$ nerovnost).

Lze použít pro komparační funkci.

56. Komparační funkce

Vzdálenost p_i od p:

```
def dist(p1, p2):                          #Compute distance
    dx = p1.x - p2.x
    dy = p1.y - p2.y
    return sqrt(dx * dx + dy * dy)
```

Vzdálenost p_i od T:

```
def getT(points):                          #Compute center of mass
    xt = 0; yt = 0
    n = len(points)
    for pi in points:
        xt += pi.x
        yt += pi.y
    return Point(xt/n, yt/n)               #Return its coordinates
```

Použití komparátoru:

```
p = Point(0, 0)
points.sort(key= lambda k: dist(k, p))
T = getT(points)
points.sort(key= lambda k: dist(k, T))
```

57. Objektově orientovaný návrh programu

Vychází z objektového přístupu k řešení problému.

Vytváří objektový model problému.

Na základě analýzy objektového modelu následně navrhovány a používány techniky vedoucí k efektivnímu řešení problému.

Postup je tvořen několika kroky:

1. Formulace problému.
2. Návrh algoritmu.
3. Předběžný návrh architektury.
4. Podrobný návrh architektury.

58. Formulace problému

Formulace problému, jehož řešení hledáme.

Nevhodná formulace může výrazně ztížit proces nalezení řešení.
Popř., takové řešení nemusí existovat.

Definujeme:

- Požadavky nutné pro řešení problému.
- Formu výsledku.

Na tomto základě volíme další postup spočívající v:

1. návrhu algoritmu,
2. jeho implementaci.

59. Návrh algoritmu

Proveden na základě dvou faktorů:

- **Typu řešeného problému:**
  Existence exaktního řešení.
  Požadavek přesného nebo přibližného řešení.
  Možnost dekompozice problému.
  Velikost množiny vstupních dat.
  Přesnost vstupních dat.

- **Požadavku na výsledky:**
  Přesnost řešení (chyba modelu, chyba výstupních dat).
  Časová složitost.
  Paměťová složitost.
  Postup nazýváme **strategií implementace**.

60. Předběžný návrh architektury

Zahrnuje:

- úvahu o rozčlenění problému na třídy,
- operace, které mohou být s jejich instancemi prováděny,
- návrh rozhraní umožňujícího komunikacemi mezi instancemi.

Grafický návrh:

V této fázi je vhodné použít grafický návrh.

Ilustruje závislost mezi jednotlivými komponentami formou diagramu či tabulky.

Výhoda grafického návrhu:

Umožní lépe pochopit vazby a vztahy mezi jednotlivými komponentami.

61. Podrobný návrh architektury

Strategie zjemňování:

Návrh opakovaně upravujeme a zpřesňujeme.

Postup představuje aplikaci postupu shora dolů (Top-Bottom),

Při návrhu zacházíme do čím dál hlubších podrobností.

Případné nedostatky nebo neefektivní konstrukce nahrazujeme vhodnějšími postupy.

3 fáze:

- Návrhy tříd.
- Návrhy datových položek.
- Návrhy metod.

62. Postup návrhu tříd

1. Stanovíme jednotlivé třídy, které budou sloužit k řešení dílčích podproblémů.
2. Míra konkrétnosti podproblému:
   Třídy pro práci s daty (ukládání či načítání),
   Třídy realizující výpočty,
   Třídy zajišťující interakci s uživatelem,
   Třídy poskytující grafické rozhraní (lepší čitelnosti a udržovatelnosti programu).

Komplexní návrh tříd představuje složitý problém.

V průběhu tohoto procesu často zjišťujeme, že:

- některé třídy mohou konat duplicitní činnosti,
- existují činnosti, které nejsou vykonávány žádnou třídou.

Musíme proto postup několikrát opakovat, než vznikne použitelnější varianta návrhu.

63. Návrhy datových položek

Návrh způsobu, jakým budou uložena data,

Ovlivňuje efektivitu operací prováděnými s těmito daty.

Nutno zohlednit několik faktorů:

- Volbu základních či uživatelských datových typů.
- Volbu vhodných dynamických datových struktur (seznam, fronta, zásobník) vzhledem ke způsobu řešení problému (přímý či sekvenční přístup).
- Volbu vhodných datových typů vzhledem k požadované přesnosti vstupních a výstupních dat.
  Poznámka: výpočty s vysokou přesností: `double` nebo `long`.

64. Návrh metod

Metody popisují činnosti, které mohou být s daty prováděny.

V tomto kroku navrhujeme rozhraní.

Stanovujeme, které z metod mají být veřejné, a které soukromé.

Výsledkem návrhu tabulka objektů a jejich metod.

Znázorňuje operace prováděné s jednotlivými objekty a jejich hierarchii.

Zásada:

- S každým objektem by měla být prováděna alespoň jedna operace ,
- Každé operaci by měl odpovídat nějaký objekt, se kterým bude prováděna.

Pokud se vyskytnou metody a objekty, které tuto podmínku nesplňují, je nutno provést korekci návrhu.

65. Návrh metod

Vlastní návrh metod:

1. Volba formálních parametrů metod.
2. Volba datových typů formálních parametrů metod.
3. Volba návratových typů metod.

Nutno zohlednit požadovanou přesnost dat !!!

Zásada přesnosti prováděných operací:

Přesnost operací prováděných s daty uvnitř jednotlivých metod musí být stejná nebo vyšší než přesnost, s jakou jsou data uložena.

V opačném případě by mohlo dojít ke ztrátě přesnosti a nežádoucímu ovlivnění výsledků.