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

Metodický přístup zahrnující popis a řešení problému s využitím tzv. objektového programování => paradigma OOP.

V současné době jeden z nejčastěji používaných přístupů, existuje ve velké většině programovacích jazyků.

Charakteristika OOP

- Vychází z objektového popisu řešeného problému, který je abstrakcí skutečného problému.
- Poskytuje vhodné nástroje pro řešení tohoto problému.
- Není vázáno tak silně na algoritmus jako procedurální přístup.

Řešení problému s využitím OOP je více-méně **analogií** k postupu, kterým by tento problém řešil člověk.

Postup řešení je v řadě případů efektivnější než při procedurálním přístupu (ale ne vždy!).

Využívá konstrukce známé z procedurálního programování (funkce, proměnná), které skládá 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) tak, aby byl 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 jsou 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 nebo zasílá požadavky druhým objektům, a to ve formě zpráv (prováděné operace, výjimky).

Objekty se liší svým **vnitřním stavem**, který se mění v průběhu vykonávání programu.

3. Struktura objektu

Objekty jsou kombinací dat a funkcí, které nad těmito daty pracují (známy z procedurálního programování).

Objekt tvořen:

- Datovými strukturami.
- Metodami.

Datová struktura:

Datová struktura ovlivňuje vlastnosti objektu.

Představována proměnnými různých datových typů.

Tato část objektu je soukromá, z vnějšku je před jinými objekty ukryta => *princip zapouzdření*.

Metody:

Metody určují chování objektu.

Definují operace, které je možno s daty provádět.

Tato část objektu je veřejná.

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

4. Platnost a životnost objektů

Platnost objektu x životnost objektu:

Část programu, ve kterém lze objekt použít x Časový interval, po který může být objekt při běhu programu použit.

Objekt chápán jako *lokální proměnná* (alokován na zásobníku) či alokován dynamicky (na haldě).

Paměť přidělována na žádost operačním systémem (ne kompilátorem), nepotřebná paměť není uvolňována automaticky.

Možnost vytváření datových struktur za běhu programu.

Některé jazyky (Java, Python) disponují automatickou správou paměti.

Ta je realizována tzv. *garbage collectorem*.

Garbage collector:

Provádí automatické rušení takových objektů, na které neexistuje žádný odkaz.

Programátorovi odpadá práce spojená s ručním rušením objektu.

Nevýhodou tohoto postupu jsou vyšší paměťové nároky na běh aplikace.

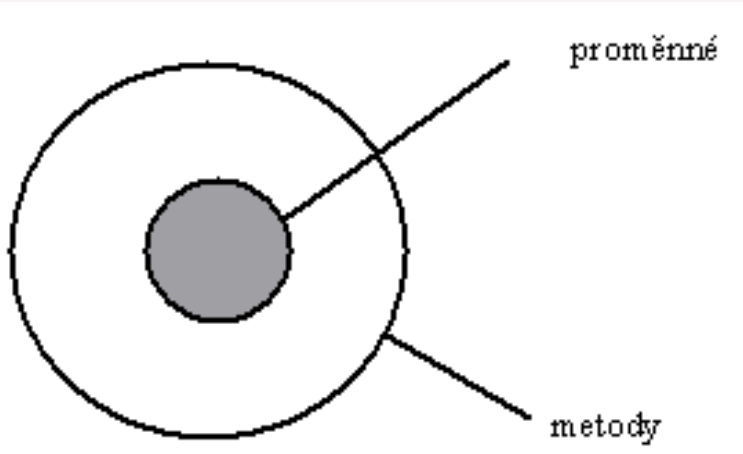
5. Princip zapouzdření

Zapouzdření = Encapsulation.

Zapouzdření umožňuje před ostatními objekty **zatajit** vnitřní stav objektu.

Při náhodných operacích nemohou jiné objekty měnit stav objektu přímo a zanést do něj nechtěné chyby.

Taková nechtěná změna by mohla ovlivnit funkčnost jiné části programu.



Analogie s ovocem, jádro tvoří data, slupku metody.

Metody umožňují objektu komunikovat se svým okolím, představují jeho rozhraní.

Proměnné instance nejsou z vnějšku přímo viditelné, nemůžeme s nimi ani přímo manipulovat.

K tomuto účelu lze využít pouze metody.

6. Zapouzdření a jeho výhody

Výhody zapouzdření:

- *Zatajování informací*
  Objekt má své rozhraní, které mu umožňuje komunikovat s ostatními objekty (chceme -li pracovat s objektem, musíme znát jeho rozhraní). Proměnné instance nejsou jinému objektu přístupné z důvodu možného zavlečení chyby.
- *Modularita*
  Každý objekt lze udržovat a spravovat nezávisle na jiném objektu, aniž by to nějak ovlivnilo celkovou funkčnost programu.

Zveřejňování rozhraní není příliš častým případem, jedná se spíše o výjimku používanou ve speciálních případech.

Vede k popření zásad a přístupu objektově orientovaného programování.

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.
3. *Používání rozhraní pro vzájemnou komunikaci objektů.*
   Důsledek bodů 1, 2.
4. *Opatrné využívání dědičnosti:*
   Může porušit zapouzdření.

8. Komunikace objektů

Objekty spolu komunikují prostřednictvím zpráv.

K tomuto účelu používány metody.

Pokud objekt A vyžaduje po objektu B, aby vykonal nějakou činnost, zašle mu zprávu v následujícím tvaru:

1. Objekt, na kterém se má akce provést
2. Činnost, která bude vykonána pomocí metody
3. Seznam parametrů předávaných metodě

Ve zprávě není obsažena informace jak tuto činnost provést (způsob implementace nemusí být znám), ale pouze co provést.

Režie s vykonáním činnosti je dána objektu.

9. Třída (Class)

Třída je šablona sloužící k definici uživatelských datových typů.

Třída představuje abstrakci chování a vlastností skutečných objektů.

Tyto datové typy bývají proto nazývány jako **abstraktní datové typy (ADT).**

Objekty tvoří *instance* těchto datových typů.

Jeden objekt nezávislý na druhém.

Třída určuje, jak bude objekt vypadat, a jak se bude chovat.

Třída je tvořena komponentami nazývanými **členy třídy**, představují je datové položky a metody.

10. Datové položky třídy a instance

Datové položky instance.

Z jedné třídy můžeme vytvořit libovolný počet objektů, každý z nich má k dispozici svojí vlastní sadu datových položek nazývaných také *proměnné instance*.

Datové položky instance jsou iniciovány při vytvoření instance, existují po celou dobu životnosti instance.

Proměnné jedné instance jsou *nezávislé* na proměnných jiné instance.

Datové položky třídy.

Datové položky třídy jsou společné všem instancím vytvořeným z jedné třídy.

Nejsou vázány na konkrétní instanci, jsou společné všem instancím třídy, bývají nazývány jako *statické proměnné.*

V programech mají speciální funkce, používají se pro sledování stavu instancí, např. jejich počtu.

11. Metody třídy a instance

Metody instance:

Metody instance voláme vždy pro konkrétní instance.

Pracuje s proměnnými instance i s proměnnými třídy.

Představuje zprávu poslanou konkrétní instanci.

Lze je volat až v případě, kdy je v programu vytvořena konkrétní instance.

Metody třídy:

Nejsou volány pro konkrétní instanci, představují zprávu zaslanou třídě jako celku.

Smějí pracovat pouze s proměnnými třídy, nikoliv s proměnnými instance, bývají nazývány statickými metodami.

Používány ve speciálních případech, např. pro sledování stavu instancí (vrácení počtu vytvořených instancí konkrétní třídy).

12. Přístup ke členům třídy

Využíván princip zapouzdření, některé členy třídy označeny jako soukromé, jiné jako veřejné.

Modifikátory přístupu:

Řídí přístup ke členům třídy.

Programátor může ovlivnit, ke kterým částem objektu bude možno přistupovat z vnějšku a kterým nikoliv, které části objektu bude modifikovat a jak.

Modifikátor `public`: přístup z libovolní třídy.

Modifikátor `private`: přístup pouze z třídy, kde byly deklarovány.

Modifikátor `protected`: přístup ze třídy, kde byly deklarovány + přístup z odvozených tříd.

Soukromé členy třídy deklarovány zpravidla jako `private`, veřejné členy třídy jako `public` , v kombianaci dědičnosti `protected`.

13. Příklad 1: Návrh třídy Auto

Návrh třídy vyplývá z požadované funkčnosti třídy (tj. co má třída dělat), zahrnuje:

- Vytvoření zjednodušeného objektového modelu, který je abstrakcí skutečného objektu.
- Na jeho podkladě provádíme volbu datových položek (vlastnosti) a nastavení jejich funcionality.

Cíl: Navržení třídy `Auto`, která představuje abstrakci skutečného automobilu.

Možná kritéria ovlivňující vlastnosti automobilut: hmotnost, okamžitá rychlost, maximální rychlost, rok výroby, délka, stav.

Datové položky instance: `hmotnost`, `o_rychlost`, `m_rychlost`, `rok_vyroby`, `stav`.

Chování automobilu je dáno vlastnostmi: zabrzdi, přidej plyn, jsi na opravu, jdi do šrotu :-).

Metody instance: `zabrzdi()`, `pridej()`, `oprava()`, `srot()`.

14. Konstruktor třídy

Z vnějšku k datovým položkám nemůžeme přistupovat, jsou privátní!!!
Datové položky třídy se inicializují v tzv. *konstruktoru třídy*.

Konstruktor třídy:
Představuje speciální metodu volanou při vytvoření objektu.
Lze ho přetěžovat.
Název je v řadě programovacích jazyků shodný se jménem třídy.
Nemá návratovou hodnotu (ani void).

Typy konstruktorů:
2 typy konstruktorů:

- *Bezparametrický konstruktor*
  Nepředáváme mu žádné informace zvnějšku o datových položkách třídy.
  Všechny instance vytvořené tímto způsobem mají stejné vlastnosti.
- *Parametrický konstruktor*
  Lze mu předávat zvnějšku informace o hodnotách datových položek. Každá instance vytvořená s použitím parametrického konstruktoru může mít jiné vlastnosti.

15. Příklad 1: Návrh třídy Auto

```
public class Auto {

  private static int pocet=0; //Staticka promenna
  private int rok_vyroby; //Promenne instance
  private boolean stav;
  private float hmotnost, o_rychlost, m_rychlost;

  public Auto() //Bezparametricky konstruktor
  {
    pocet++; //Inkrementace pri vytvoreni noveho objektu
    rok_vyroby = 2000;
    stav = true;
    hmotnost = 1000;
    o_rychlost = 100;
    m_rychlost = 130;
 }
```

16. Příklad 1: Návrh třídy Auto

```
//Parametricky konstruktor
public Auto(int rok_vyroby_, boolean stav_, float hmotnost_,
   float o_rychlost_, float m_rychlost_)  {
     pocet++;
     rok_vyroby = rok_vyroby_;
     stav = stav_;
     hmotnost = hmotnost_;
     o_rychlost = o_rychlost_;
     m_rychlost = m_rychlost_;
 }
```

Nebo

```
public Auto(int rok_vyroby_, boolean stav_, float hmotnost_,
   float o_rychlost_, float m_rychlost_) : rok_vyroby(rok_vyroby_),
   stav(stav_), hmotnost(hmotnost_), o_rychlost(o_rychlost_),
   m_rychlost(m_rychlost_) {pocet++;}
```

17. Zásady OOP:

Základní zásady OOP:

- Zapoudření (Encapsulation).
- Abstrakce (Abstraction).
- Kompozice - opětovné použití implementace (Composition)
- Dědičnost - opětovné použití rozhraní (Inheritance).
- Polymorfismus (Polymorfism).

18. Abstrakce

Zobecnění:

Představuje nalezení společného, nadřazeného, zastřešujícího významu či pojmu pro skupinu prvků.

Příklady zobecnění:

- Mrkev, kedlubna, květák, cibule, pórek ⇒zelenina.
- Zelenina, ovoce, obilniny ⇒potraviny.

Existuje několik úrovní zobecnění lišící se mírou podrobnosti zobecnění.

Abstrakce:

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

Abstrakce představuje oprostění se od detailů, které nemusejí být podstatné.

Příklad abstrakce:

Švestka, třešeň, jablko, kokosový ořech, ananas ⇒ovoce.

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

Zanedbávány mají být pouze méně důležité vlastnosti ⇒ tj. abstrahujeme se od nich.

19. Abstrakce a programování

Konceptuální zjednodušení problému zanedbáním méně podstatných či nepodstatných detailů, vlastností.

Vlastnosti a funkce skutečného objektu přenášeny na nehmotný objekt.

Tento přístup návrhu řešení problému umožňuje programátorovi abstrahovat se od detailů, jakým způsobem vnitřně pracují jednotlivé objekty.

Pro práci s objektem mu postačuje znalost rozhraní, kterým objekt komunikuje se svým okolím.

Programátor objektu sděluje, co má udělat, a nikoliv jak to má udělat.

Důsledky zobecnění a abstrakce pro programování:

Hledání společných vlastností, zastřešujících nadtříd je základním rysem objektového návrhu programu.

20. 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).
4. Dodržovat jednotnou úroveň abstrakce komponent.

Rozčlenění problému do tříd, výhody:

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

21. Kompozice

Složité objekty chápány jako kolekce objektů jednodušších.

Objekty mohou být skládány (komponovány) z jednodušších objektů, akce mohou být redukovány na jednodušší.

Postup se opakuje tak dlouho, dokud objekty i akce nejsou triviální (tj. snadno použitelné a řešitelné).

Kompozice je jedna ze základních vlastností OOP, výsledkem je znovupoužitelný kód.

Kompozice (skládání) představuje použití instance jedné třídy v definici druhé třídy.

Třídu lze "skládat" z více objektů jiných tříd.

Kompozice umožňuje použití hierarchického přístupu při návrhu tříd.

Kompozice představuje přístup, při kterém z jednodušších komponent skládáme složitější komponenty.

Agregace: agregované části objektu mohou existovat i bez objektu.

Kompozice: kompozity nemohou existovat bez objektu.

22. 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 společné vlastnosti a chování od objektů, ze kterých jsou odvozeny.

Dědičnost představuje *opětovné použití rozhraní*, umožňuje používat při návrhu tříd hierarchický přístup.

Princip dědičnosti:

Třídy na nižší úrovni mohou dědit vlastnosti a chování tříd nacházejících se na vyšší úrovni.

Novou třídu můžeme odvodit z podobné existující třídy přidáním požadovaných funkcí a vlastností.

V užším slova smyslu se pro spojení dědičnosti a polymorfismu používá termín *objektově orientované programování*.

Původní třída = rodičovská, bázová třída.

Odvozená třída = třída potomků.

23. Předek, potomek

Předek a potomek:

Děděním vzniká z předka potomek.

Potomek zdědí od předka všechny vlastnosti, zpravidla mu definujeme některé další vlastnosti a metody, které jsou pro něj specifické.

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

Platí zásada, že potomek může vždy zastoupit předka.

Dědická hierarchie:

Posloupnost rodičů a jejich potomků vytváří dědickou hierarchii.

Lze ji znázornit stromovou strukturou.

Předefinování metody:

Metoda rodičovské může být v odvozené třídě předefinována a vybavíme ji tak novou funkčností. Předefinovaná metoda má stejnou hlavičku, liší se pouze “výkonnou” částí.

24. Dědičnost, výhody, použití

Dědit lze jak rozhraní, tak implementaci.

Při dědění *rozhraní* není děděn kód, ale pouze signatura rozhraní (prototypy metod a datové položky).

V odvozené třídě dopsána 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, třídy `final`.

25. Substituční princip

tzv. Liskov Substitution Principle (LSP).

Řeší vzájemný vztah mezi dvěma odvozenými třídami.

Zásady:

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

26. Dědičnost vs. kompozice (agregace)

ISA-HAS test (testování první zásady LSP), 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 `Bod` obsahuje dvě datové položky x,y představující souřadnice bodu.

Třída `Linie` bude obsahuje dvě datové položky typu Bod.

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

Dotaz, zda je linie bod, postrádá smysl.

27. Příklad: Ukázka kompozice

```
public class Bod {
  private int double x,y;
  public Bod(double x, double y)  {
     this.x=x;
     this.y=y;
  }

  public double getX() {
     return this.x
  }
  public double getY() {
     return this.y
  }
}
```

28. Příklad: Ukázka kompozice

```
public class Usecka
{
  private Bod start, end; //Kompozice

  public void (Bod start, Bod end) {
      this.start.x=start.x;
      this.start.y=start.y;
      this.end.x=end.x;
      this.end.y=end.y;
 }
 public void Vypis() {  //Vypis souradnic koncovych bodu
      System.out.println(this.start.getX());
      System.out.println(this.start.getY());
      System.out.println(this.end.getX());
      System.out.println(this.end.getY());
 }
```

29. Příklad: Ukázka kompozice

```
public class Main() {
  public static void main(String [] args)  {
      Bod s=new Bod(0,0);   //Pocatecni bod
      Bod k=new Bod(10,10); //Koncovy bod

      Usecka u=new(s,k);    //Usecka

      u.vypis();  //Vypis usecky
 }
}
```

30. Příklad: Dědičnost, návrh odvozené třídy Osobni

Z třídy `Auto` odvodíme novou třídu `Osobni`.

Odvozené třídě přidáme novou funkcionalitu vyjádřenou datovými položkami a metodami.

Novými datové položky: maximální počet lidí, které lze v osobním autě přepravit, aktuální počet lidí a velikost zavazadlového prostoru.

Proměnné instance: `max_pocet_lidi`, `pocet_lidi`, `z_prostor`.

Nové metody: přistoupení spolujezdce, vystoupení spolujezdce, naložení a vyložení zavazadlového prostoru.

Metody instance: `pristup()`, `vystup()`, `naloz()`, `vyloz()`.

31. Příklad: Návrh odvozené třídy Osobni

```
public class Osobni extends Auto {
   private int pocet_lidi, max_pocetlidi;
   private float z_prostor;

   public Osobni(int rok_vyroby, boolean stav, float hmotnost,
        float o_rychlost, float m_rychlost, int pocet_lidi,
        int max_pocetlidi, float z_prostor) {

      //Volani konstruktoru rodicovske tridy
      super(rok_vyroby,stav,hmotnost,o_rychlost, m_rychlost);

      //Inicializace datovych polozek odvozene tridy
      this.pocet_lidi=pocet_lidi;
      this.max_pocetlidi=max_pocetlidi;
      this.z_prostor=z_prostor;
   }
```

32. Příklad: Návrh odvozené třídy Osobni

Ukázky metod odvozené třídy Osobni:

```
public void pristup(int kolik) {
    this.pocet_lidi+=kolik;
}

public void vystup(int kolik) {
    this.pocet_lidi-=kolik;
}

public void naloz(float kg) {
    this.z_prostor+=kg;
}

public void vyloz(float kg){
    this.z_prostor-=kg;
}
```

33. Konstruktory odvozené třídy

Hierarchie volání konstruktorů:

- V těle konstruktoru odvozené třídy volán konstruktor *rodičovské* třídy, který se postará o inicializaci datových položek rodičovské třídy.
  Definici datových položek odvozené třídy může záviset na definici datových položek rodičovské třídy !!!
- Teprve následně jsou inicializovány datové položky třídy *odvozené*.

Při vytvoření instance odvozené třídy nejprve v těle konstruktoru volán konstruktor rodičovské třídy a teprve následně je provedeno tělo konstruktoru odvozené třídy.

```
//Volani konstruktoru rodicovske tridy
super(rok_vyroby,stav,hmotnost,o_rychlost, m_rychlost);
```

34. Finální třídy a finální metody

Finální třídy nelze dědit, označovány jako koncové třídy $\Rightarrow$ zákaz dědičnosti.

Deklarovány s klíčovým slovem `final`, vyšší rychlost při překladu (optimalizace).

```
final class Auto {
    ....
}

class Osobni extends Auto //Nelze odvodit od finalni tridy
```

Finální metody nelze v odvozené třídě předefinovat.

Důvody použití: zaručení stejné funkčnosti metody i v odvozené třídě.

```
class Auto{
    public final void zabrzdi();
}
class Osobni extends Auto{
    public void zabrzdi();  //Nelze predefinovat
}
```

Označení metody jako finální nevynucuje označení třídy jako finální, naopak neplatí.

35. Abstraktní třídy a abstraktní metody

Abstraktní metody musí být v odvozené třídě předefinovány.

Protiklady finálních metod.

Abstraktní metody se mohou vyskytovat pouze v abstraktních třídách.

Abstraktní metody nemají žádný výkonný kód.

Tímto postupem vynutíme předefinování metody.

```
abstract class Auto {
   abstract void zabrzdi();  //Musi byt predefinovana v odv
}
class Osobni extends Auto{
   void zabrzdi() {
      //Nejaky kod
   }
}
```

Použití u knihoven.

36. 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), vede k nepřehlednosti.
- Zamyslet se nad použitím třídy s jedním potomkem (má smysl?).

37. Dědičnost / kompozice a problémy

Možné problémy a úskalí kompozice / dědičnosti:

- Přílišné nadužívání kompozice i dědičnosti v případech, kdy to není potřeba (příliš složitý návrh).
- 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, zvláště ve spolupráci s `protected` položkami.

38. Polymorfismus

Polymorfismus (mnohotvarost) způsobuje že objekty na stejný podnět reagují různě, a to v závislosti na svém typu.

Každá operace může mít více implementací, konkrétní implementace zvolena v závislosti na typu objektu, s nímž se operace provádí.

Použití polymorfizmu:

- *Přetěžování funkcí / metod*
  Stejný identifikátor, liší se počtem / typem parametrů.
- *Dědičnost rozhraní / implementace*
  Rodičovská třída použita k definici rozhraní (společné vlastnosti), odvozené třídy implementují odlišnosti.

Pro implementaci polymorfismu je používána pozdní vazba (v C++ realizována virtuálními metodami).

39. Polymorfismus, statická a dynamická vazba.

Důsledek polymorfismu:

Polymorfismus (mnohotvarost) způsobuje že instance různých tříd v jedné dědické hierarchii na stejný podnět reagují různě.

V místě, kde je očekávána instance nějaké třídy, můžeme použít i instanci její libovolné odvozené třídy odvozené přímo či nepřímo.

Potomek může zastoupit předka.

Tato instance se může chovat jinak, než by se chovala instance rodičovské třídy (pouze v rozsahu daném popisem rozhraním).

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

Při překladu programu je známo, zda bude použita metoda předka či potomka.

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

O tom, zda bude použita metoda předka či potomka, rozhodnuto až v době běhu programu (virtuální metody).

Java používá pouze pozdní vazbu. C++ nutné klíčové slovo virtual.

40. Ukázka polymorfismu

Odkaz ukazuje na stejný typ objektu, jakého je sám typu:

```
Auto a = new Auto();
Osobni o = new Osobni();
a.zjistiVlastnosti();  //Vrati vlastnosti auta
o.zjistiVlastnosti();  //Vrati vlastnosti osobniho auta
```

Odkaz ukazuje na jiný typ objektu (v kontextu dědické posloupnosti), jakého je sám typu:

```
Auto c;
c = a;                  //Ukazuje na potomka
c.zjistiVlasnosti();  //Volani metody tridy Auto
c = o;
c.zjistiVlasnosti();  //Volani metody tridy Osobni
```

Statická vazba: volána metoda dle datového typu odkazu (překladač nezajímá typ objektu, kam odkaz ukazuje).

Dynamická vazby: volána metoda dle datového typu objektu, na který odkaz ukazuje.

41. Přetypování předek $\Longleftrightarrow$potomek.

Přetypování potomka na předka (upcasting):

Potomka na předka lze přetypovat automaticky, tzv. *implicitní přetypování*.

Potomek umí více než předek

Dochází k funkční degradaci („skrytí” všech metod odvoz. třídy).

Objekt `a` typu `Osobni`, ale chová se jako *`Auto`*

```
Auto a = new Auto();
Osobni o = new Osobni();
a = o;  //Pretypovani potomka na predka...
```

Přetypování předka na potomka (downcasting):

Předka na potomka lze přetypovat pouze *explicitně*.

Předek ale umí méně než potomek.

Zpětné přiřazení může způsobit, že po potomkovi budeme požadovat činnost, kterou neumí.

```
o = a;          //Nelze, implicitni pretypovani
o = (Osobni)a; //OK, explicitni pretypovani
```

42. Hluboká a mělká kopie

Mělká kopie objektu:

Kopírují se pouze hodnoty atributů.

Pokud je nějaký atribut objektového typu (tj. odkaz na objekt), kopíruje se pouze odkaz na tento objekt.

Nejsou prováděny žádné alokace paměti.

Kopie je závislá na originálu (kopírování odkazů).

Hluboká kopie objektu:

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).

43. Kopírování, příklad

```
public class Trida {
    double n;
    double [] pole;

    Trida ( int n) {
        this.n=n;
        pole=new double[n];
        for (int i=0;i<n;i++) pole[i]=n;
    }
}

Trida t1 = new Trida(5);
Trida t2 = t1; //Zkopírování odkazu
```

44. Metoda clone

Realizuje kopírování objektů v Javě.

V C++ používán tzv. kopírovací konstruktor.

Metoda `clone()` může vytvářet jak mělkou, tak hlubokou kopii.

Postup:

1. Implementace rozhraní `Cloneable` třídou.
2. Předefinování protected metody `clone()`, návratový typ shodný se třídou.
3. Výkonný kód ukryt v bloku `try/catch`.
4. Nový objekt není vytvořen pomocí kostrukce `super.clone()`.
5. Zkopírování jednotlivých atributů.
6. Vrácení kopie objektu.

45. Ukázka mělké kopie

```
protected Object clone() {
      Trida kopie = null;
      try
      {
          kopie = (Trida)super.clone();
          kopie.n = this.n;
          kopie.pole = this.pole;
      }
      catch (CloneNotSupportedException e)
      {
          e.printStackTrace();
      }
  return kopie;
  }

Trida t3 = (Trida)t1.clone(); //Vytvoreni melke kopie
```

Nutno ošetřit zkopírování prvků pole;

46. Ukázka hluboké kopie

```
protected Object clone(){
        Trida kopie=null;
        try
        {
            kopie=(Trida)super.clone();
            kopie.n=this.n;
            kopie.pole=this.pole.clone();  //Kopie pole
        }
        catch (CloneNotSupportedException e)
        {
            e.printStackTrace();
        }
    return kopie;
    }

Trida t3=(Trida)t1.clone(); //Vytvoreni hluboke kopie
```

47. Porovnání dvou instancí

Třída představuje definici vlastního uživatelského typu.

Jak porovnat dva uživatelské datové typy?

```
class point{
   double x, y;
   public Point(double x, double y){
     this.x=x;this.y=y;
   }
}
```

Vytvoříme -li dva body

```
Point p1=new Point(0,0);
Point p2=new Point(10,10);
```

Který z bodů je „větší”/”menší”!?

Nelze jednoznačně určit, nutnost definice komparátoru (porovnávače).

48. Komparátor a jeho vytvoření

Implementuje rozhraní `Comparable`.

Důležitá metoda `CompareTo()` porovnávající datové položky třídy.

Návratová hodnota typu int >0, =0, <0 vyjadřující relaci.

```
public class Point implements Comparable<Point>{
    private double x,y;

    public Point(double x, double y){this.x=x;this.y=y;}
    public double getX(){return this.x;}
    public double getY(){return this.y;}

    //Setrideni podle x
    public int  compareTo(Point p) {
        if (this.x < p.x) return 1;
        else if (this.x > p.x) return -1;
        else return 0;
    }
```

49. Komparátor a jeho vytvoření

```
 //Setrideni podle y
public int  compareTo(Point p) {
     if (this.y < p.y) return 1;
     else if (this.y > p.y) return 0;

//Setrideni podle vzdalenosti od pocatku
public int  compareTo(Point p) {
     double d1 = this.x * this.x + this.y * this.y; //Kvadrat
     double d2 = p.x*p.x + p.y*p.y; //Kvadrat
     if (d1 < d2) return 1;
     if (d1 > d2) return -1;
     else return 0;
 }
```

Komparátor nemůžeme přetěžovat.

50. Setřídění seznamu bodů

Vytvoření nového seznamu bodů:

```
List <Point> points=new ArrayList<Point>();
```

Přidání bodů do seznamu:

```
points.add(new Point(0,0));
points.add(new Point(10,10));
```

Vlastní setřídění:

```
Collections.sort(points);  //Setrideni internim sortem
```

51. Vytvoření komparační třídy

Další možnost definující vztahy mezi datovými položkami třídy.

Univerzálnější, lze porovnávat i dle jiných než číselných typů (String).

Třída implementuje rozhraní `Comparable`, předefinováváme v ní metodu `compare()`.

```
public class sortPoints implements Comparator<Point>{
    public int compare(Point p1, Point p2) {
        return p1.getX.compareTo(p2.getX());
    }
}
```

Při třídění vytváříme novou instanci této třídy:

```
Collections.sort(points, new sortPoints());
```

52. 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.

53. 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 návrhu algoritmu a jeho implementaci.

54. 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**.

55. 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, který ukazuje závislost mezi jednotlivými komponentami formou diagramu či tabulky.

Výhoda grafického návrhu:

Umožní lépe pochopit vazby a vztahy mezi jednotlivými komponentami.

56. Podrobný návrh architektury

Strategie zjemňování:

Návrh opakovaně upravujeme a zpřesňujeme.

Postup představuje aplikaci postupu shora, 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.

57. 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 nebo naopak existují činnosti, které nejsou vykonávány žádnou třídou.

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

58. 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`.

59. Návrh metod

Metody popisují činnosti, které mohou být s daty prováděny.

V tomto kroku navrhujeme rozhraní a stanovujeme, které z metod mají být veřejné, a které soukromé.

Výsledkem návrhu tabulka objektů a jejich metod znázorňující 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.

60. 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.