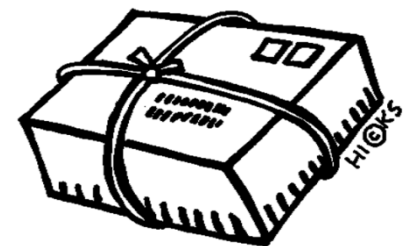




11. prednáška (30.11.2011)

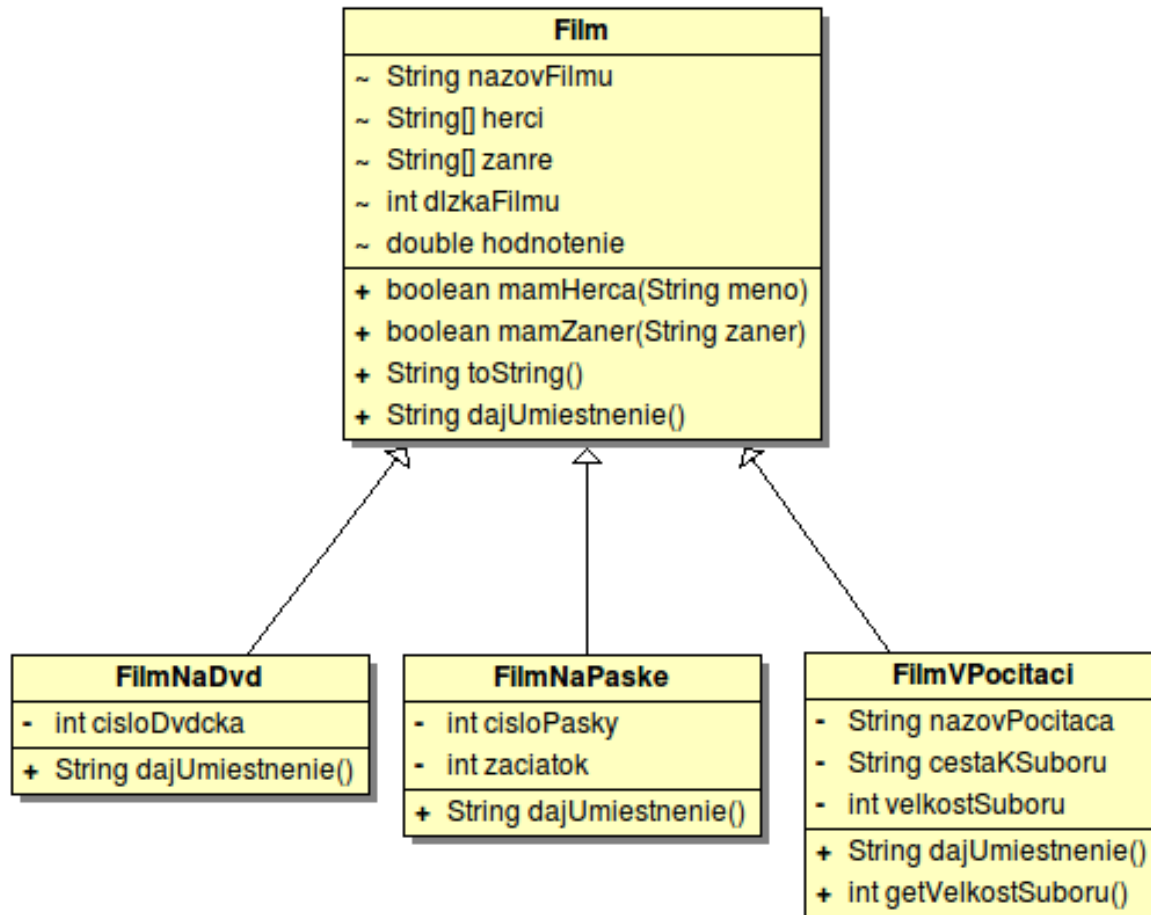
**Abstrakné triedy a
metódy, interface-y,
triedenie, balíky,
modifikátory
viditeľnosti**





Krátke zopakovanie

- Vytvorili sme si pomocou dedičnosti a polymorfizmu zoznam filmov, ktorý mal v sebe objekty tried `FilmNaDvd`, `FilmNaPaske` a `FilmVPocitaci`, ktoré dedili od triedy `Film`





Krátke zopakovanie

- Povedali sme si, že z premennej predka môžeme referencovať potomkov
 - `Film f = new FilmNaDvd();`
 - Typ premennej určuje, čo vieme na referencovaných objektoch volať
 - Z premennej `f` môžeme volať metódy definované v triede `Film`
 - Metódy, ktoré voláme cez premennú `f`, spúšťa objekt
 - Objekt vie to, akej triedy je on sám a nie to, akého typu je premenná, ktorá ma uloženú referenciu naňho
 - Polymorfizmus zabezpečil, že objekt spúšťal prekryté metódy zo svojej t.j. oddedenej triedy, alebo neprekryté metódy z triedy `Film`
- Reálne sme však objekty triedy `Film` nikdy nevytvárali



Triedový diagram

objekt triedy Film:

```
nazovFilmu: "Pacho, Hybský zbojník"
herci: ["Jozef Kroner", ...]
zanre: ["komédia"]
dlzkaFilmu: 91
hodnotenie: 8,5
boolean mamHerca(String)
boolean mamZaner(String)
String toString()
String dajUmiestnenie()
```



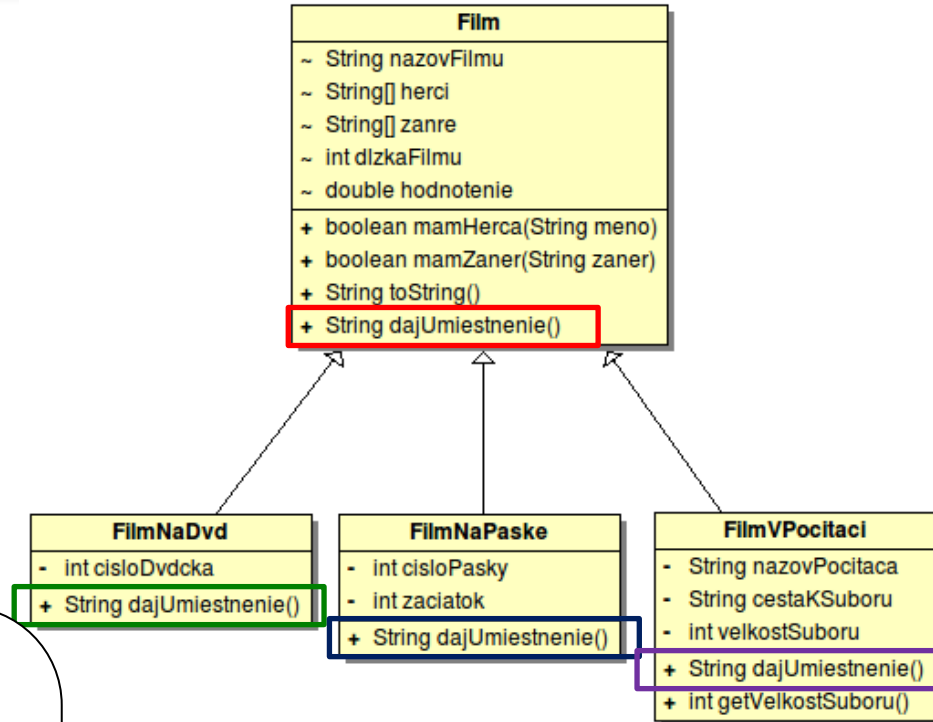
volá

objekt triedy FilmNaDvd:

```
nazovFilmu: "The Matrix"
herci: ["Keanu Reeves", ...]
zanre: ["akčný", "Sci-fi"]
dlzkaFilmu: 136
hodnotenie: 8,7
cisloDvdcka: 21
boolean mamHerca(String)
boolean mamZaner(String)
String toString()
String dajUmiestnenie()
```



volá





Prehľad prednášky

- **Abstraktné triedy**
- Abstraktné metódy
- Interfejsy (rozhrania)
- Balíčky
- Modifikátory viditeľnosti
- Triedenie



Abstraktné triedy

- Ak **chcem** aby sa objekty triedy `Film` **nikdy nevytvárali**
 - Nechcem mať film, ktorý nie je nikde uložený
- Triedu `Film` vyhlásim za abstraktnú
- Dodáme slovíčko **abstract** a všetko necháme ako pred tým

```
public abstract class Film {  
    // pôvodné telo triedy  
}
```

- Náš program funguje naďalej rovnako bezo zmeny
- Ak by niekto neskôr zabudol, a pokúšal sa vytvoriť objekt triedy `Film`, Java mu to nedovolí
- Premenné triedy `Film` môžeme naďalej deklarovať, len nikdy nebudú referencovať objekty triedy `Film`, ale len objekty oddedených tried



Prehľad prednášky

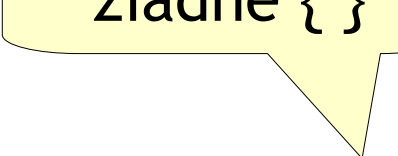
- Abstraktné triedy
- **Abstraktné metódy**
- Interfejsy (rozhrania)
- Balíčky
- Modifikátory viditeľnosti
- Triedenie



Abstraktné metódy

- Abstraktné metódy sa od normálnych metód líšia nasledovne:
 - Sú to metódy bez tela
 - Môžu sa vyskytovať len v abstraktnej triede
 - Neabstraktní potomkovia **musia** takéto metódy prekryť

```
public abstract class Film {  
    ...  
    public abstract String dajUmiestnenie();  
    ...  
}
```





Abstraktné metódy

- Ak by sme teraz vytvorili triedu `FilmNaUSB`, ktorá dedí od triedy `Film` a zabudli dodať metódu `dajUmiestnenie()` Eclipse nás upozorní:

The type `FilmNaUSB` must implement the inherited abstract method `Film.dajUmiestnenie()`

```
public class FilmNaUSB extends Film {  
    ...  
}
```



Abstraktné triedy a metódy

- Abstraktná trieda a abstraktná metóda v nej nám zabezpečia, že v poli filmov sú iba objekty filmov na nejakom médiu a každý z týchto objektov má funkčnú metódu `dajUmiestnenie()`

```
public class ZoznamFilmov {  
    ...  
    public void vypisUmiestnenia() {  
        for (int i = 0; i < filmy.length; i++) {  
            System.out.print(filmy[i].getNazovFilmu()+" : ");  
            System.out.println(filmy[i].dajUmiestnenie());  
        }  
    }  
    ...  
}
```

Všetko zbehne OK



Prehľad prednášky

- Abstraktné triedy
- Abstraktné metódy
- **Interfejsy (rozhrania)**
- Balíčky
- Modifikátory viditeľnosti
- Triedenie



Opäť zadania pre programy

- Vieme, že v každom rozumnom zadaní sa špecifikujú dve kľúčové množiny požiadaviek:
 - **S akými dátami** bude program pracovať
 - **Aké služby** má poskytovať resp. **akú funkcionality** má program mať
- Obvykle nie je povedané v akých štruktúrach máme dáta reprezentovať a uchovávať
 - Samotná realizácia je voľbou programátora
 - Reprezentáciu aj tak nevidno (privátne premenné)



Služby a nositelia dát

- Sú triedy, ktoré sú hlavne **nositelia dát**
 - u nás `Film` a jeho potomkovia
 - Medzi metódami dominujú gettery a settery
 - Nie je potrebné mať viac variantov implementácie
- Iné triedy sú zamerané na funkcionality často nazývané **služby**
 - u nás `ZoznamFilmov`
 - Môžeme chcieť viac variantov (ukladanie do textových súborov, alebo do binárnych súborov, do databázy, na server, ...; reprezentácia v poli alebo v inej štruktúre,...)
 - Funkcionalita by mala byť rovnaká pre každý variant



Kontrakt vs. reprezentácia

- Používateľa nášho programu zaujíma, či služba spĺňa požadovanú funkcionálnosť
- t.j. či je dodržaný kontrakt:
 - Metódy dávajú pre dané vstupy očakávané výstupy
 - Nie je dôležité, ako je to vo vnútri urobené
- Cudzí program je čierna skrinka, ktorá funguje tak, ako je definované v zadaní / dokumentácii



Interface ako kontrakt

- Kontrakt sa v Jave zapisuje vo forme interface-u
- Je to niečo ako „minimalistická abstraktná trieda“ bez inštančných premenných, ktorá má len abstraktné metódy (bez tela - iba hlavičky)
- `Interface` = zoznam hlavičiek metód
- O triedach, ktoré splňajú požadovanú funkcionálnosť, hovoríme, že **implementujú daný interface**



Náš zoznam filmov je jedna z možných implementácií kontraktu

- Do interfejsu spíšeme operácie, ktoré sme mali v zadaní (načítanie sa spravíme v konštruktore):

```
public interface ZoznamFilmov {
    public void vypisVsetko();
    public void vypisPodlaZanru(String zaner);
    public void vložNovyFilm(Film film);
    public void vymazFilm(String nazov);
    public void ulozSa();
}
```

žiadne inštančné
premenne

žiadne { }

nepíšeme abstract

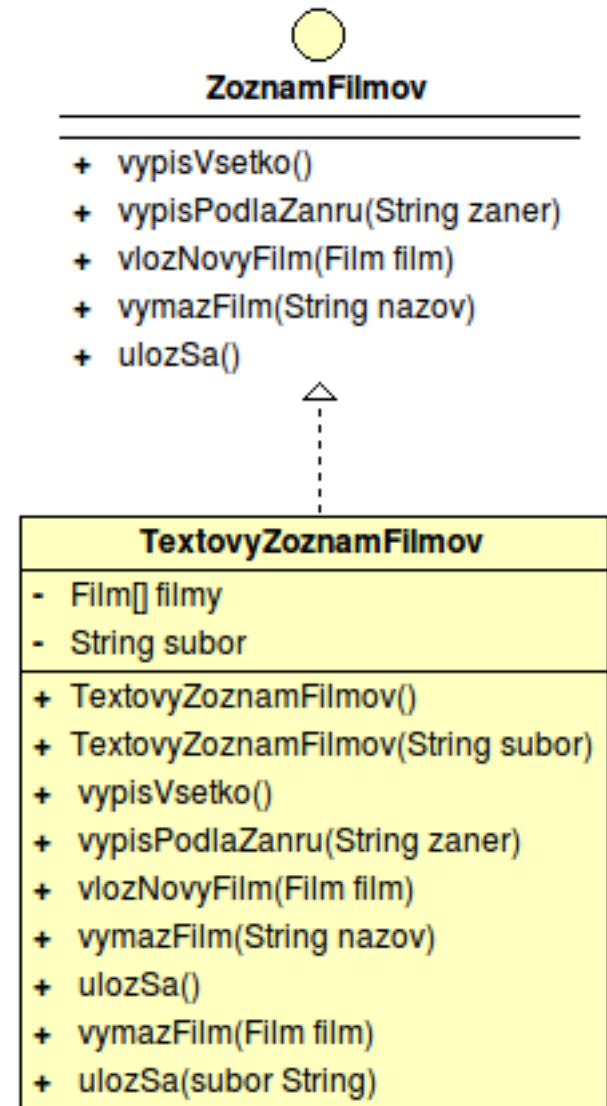


TextovyZoznamFilmov

- Náš starý ZoznamFilmov sa stane triedou TextovyZoznamFilmov implementujúcou interfejs ZoznamFilmov:

```
public class TextovyZoznamFilmov
    implements ZoznamFilmov {
    Film[] filmy = new Film[0];
    String subor = "filmy.txt";

    public void vypisVsetko() {
        ...
    }
    ...
}
```





TextovyZoznamFilmov

- TextovyZoznamFilmov **musí** implementovať všetky metódy interfejsu ZoznamFilmov, lebo by nebolo čo vykonávať pri ich volaní
- V spúšťači, ktorý zoznam filmov používa, budeme mať premennú typu ZoznamFilmov

```
ZoznamFilmov zoznamFilmov =  
    new TextovyZoznamFilmov (subor);
```

ČO chceme

AKO chceme



Iné implementácie kontraktu

- Ak chceme použiť inú implementáciu, zmeníme len jeden riadok
- Kód využívajúci zoznam filmov je stabilný, aj keď sa prehodíme z textových súborov na databázu alebo na binárne súbory

```
ZoznamFilmov zoznamFilmov =  
    new DatabazovyZoznamFilmov(url, login, heslo);
```

alebo

```
ZoznamFilmov zoznamFilmov =  
    new BinarnyZoznamFilmov("filmy.bin");
```



Interface ako rola

- Ďalším typickým použitím interfejsov sú roly
 - `Serializable` - objekt je uložitelný do postupnosti bajtov
 - `Comparable` - objekty triedy sú navzájom porovnateľné
 - `Runnable` - trieda sa vie spustiť ako samostatné vlákno
 - ...
- Využívané vo všeobecne použiteľných algoritmoch alebo štruktúrach **nad ľubovoľnými objektmi, ktoré však musia mať isté vlastnosti**, aby algoritmus mohol fungovať
 - Ukladanie objektu v binárnej forme, triedenie, spustenie metódy objektu v samostatnom vlákne,...
- Rola nepredstavuje hlavnú úlohu triedy v programe
- Trieda môže implementovať viac rolí (na rozdiel od dedičnosti)



Uletený nápad na vlastnú rolu

- Napríklad by sme si mohli vytvoriť do JPAZ-u rolu `Nakreslitelny` s metódou `nakresliSa (plocha, x, y)`
- Nápady na použitie:
 - `Lopta` extends `Turtle` implements `Nakreslitelny`
 - `Kruznica` extends `Tvar` implements `Nakreslitelny`
 - `EuroMinca` extends `Minca` implements `Nakreslitelny`
 - `Herec` implements `Nakreslitelny`
 - `FilmNaDvd` implements `Nakreslitelny`, `Spustitelny`
 - ...
- Potom by sme mohli mať pole `Nakreslitelny[]` z ktorého by sme mohli referencovať objekty typu `Lopta`, `Kruznica`, `Eurominca`, `Herec`, ... a napríklad všetky vykresliť vedľa seba, do kruhu, do štvorca, ... 😊



Prehľad prednášky

- Abstraktné triedy
- Abstraktné metódy
- Interfejsy (rozhrania)
- **Balíčky**
- Modifikátory viditeľnosti
- Triedenie



Konflikty v názvoch tried

- Doposiaľ sme mali jednoduché názvy tried - `Film`, `ZoznamFilmov`, ...
- Problém nastáva, keď dva projekty, ktoré majú figurovať v nejakom spoločnom projekte pomenujú triedu rovnako
 - `Date` - na reprezentáciu dátumu a času v Java
 - `Date` - na reprezentáciu dátumu v databáze
 - `Date` - na reprezentáciu nejakého rande v zoznamke
 - `Attribute` - spracovávač webových stránok v HTML
 - `Attribute` - v podpore pre tlač
 - `Attribute` - v projekte kapsa.sk
- Riešenie 1:
 - `JavaDate`, `SQLDate`, `RandeDate`
 - `HTMLAttribute`, `PrintAttribute`, `KapsaAttribute`
 - **Nie vždy je dohoda možná**



- Riešenie 2: Balíčky
- Zavedieme hierarchickú štruktúru podľa vzoru adresárov
 - `java.util.Date` - na reprezentáciu dátumu a času v Java
 - `java.sql.Date` - na reprezentáciu dátumu v databáze
 - `sk.rande.Date` - na reprezentáciu nejakého rande v zoznamke
 - `org.htmlparser.Attribute` - spracovávač webových stránok v HTML
 - `javax.print.attribute.Attribute` - v podpore pre tlač
 - `sk.kapsa.Attribute` - v projekte kapsa.sk



- A môžeme bez problémov pracovať s ľubovoľnou triedou:

```
org.htmlparser.Attribute atribut = new  
    org.htmlparser.Attribute();
```

- Ale nie vždy je nám to po chuti

```
org.springframework.aop  
.framework.autoproxy.metadata  
.AttributeThreadLocalTargetSourceCreator c =  
new org.springframework.aop  
.framework.autoproxy.metadata  
.AttributeThreadLocalTargetSourceCreator();
```



- Namiesto kilometrových riadkov importujeme:

```
import org.htmlparser.Attribute;
```

```
public class AttributeTester {  
    public static void main(String[] args) {  
        Attribute atribut = new Attribute();  
    }  
}
```

- Importom hovoríme, že ak píšeme `Attribute` myslíme tým `org.htmlparser.Attribute`
 - Nič viac, žiadne naťahovanie zdrojákov



- Ak chceme použiť dva rôzne typy s rovnakým menom, celému názvu sa nevyhneme:

```
import org.htmlparser.Attribute;

public class AttributeTester {
    public static void main(String[] args) {
        Attribute atribut = new Attribute();
        sk.kapsa.Attribute mojAtribut = new
            sk.kapsa.Attribute();
    }
}
```



Balíčky

- Každá trieda je v nejakom balíčku.
- Ak nemá uvedený balíček je v implicitnom balíčku (koreň hierarchie)
- Zatiaľ všetko sme mali kvôli jednoduchosti v implicitnom balíčku
- Triedy automaticky vidia iné triedy z toho istého balíčka a z balíčka `java.lang.*`, zvyšok treba importovať
- Triedy v ľubovoľnom neimplicitnom balíčku **nevidia použiť**, t.j. importovať, triedy z implicitného balíčka
- V normálnom projekte je v implicitnom balíčku **NIČ**



Balíčky

- Keďže náš projekt s filmami sa už začína podobat' na normálny projekt prehodíme ho do balíčka
- Je nepísaným pravidlom označovať prefixy balíčkov podľa inštitúcie a názvu projektu.
 - Zabránim rovnakému menu triedy aj balíčka na celom svete

```
package sk.upjs.paz.filmy;
```

```
import sk.upjs.paz.filmy.Film;
```

```
import java.io.PrintWriter;
```

```
public class FilmNaDvd extends Film {
```

```
...
```

```
}
```

netreba



- Viacero importov vieme zapísať skrátene

```
import sk.upjs.paz.filmy.Film;  
import sk.upjs.paz.filmy.FilmNaDvd;  
import sk.upjs.paz.filmy.FilmVPocitaci;  
import sk.upjs.paz.filmy.FilmNaPaske;  
  
public class Spustac {  
    ...  
}
```

```
import sk.upjs.paz.filmy.*;  
  
public class Spustac {  
    ...  
}
```

Importuj všetky
triedy z balíčka
`sk.upjs.paz.filmy`



Balíčky

- Trieda automaticky importuje len triedy zo svojho balíčka
- Ak chceme použiť triedy z nadradeného alebo podradeného balíčka, musíme importovať

```
package sk.upjs.paz;
```

```
import sk.upjs.paz.filmy.Film;
```

```
public class Spustac {  
    Film f = new Film();  
}
```

Musíme importovať



Balíčky

- V každom importe môže byť iba jedna hviezdička na konci
- Hviezdička neimportuje podbalíky
- Nasledovný zápis nezimportuje `sk.upjs.paz.filmy.Film`

```
import sk.upjs.ics.paz.*;
```

- Nevieme ani použiť:

```
import sk.upjs.ics.paz.*.*;
```




Prehľad prednášky

- Abstraktné triedy
- Abstraktné metódy
- Interfejsy (rozhrania)
- Balíčky
- **Modifikátory viditeľnosti**
- Triedenie



Modifikátory viditeľnosti

- Pomocou modifikátorov viditeľnosti vieme nastaviť viditeľnosť tried, metód a inštančných premenných
- S tým čo nevidíme, nevieme pracovať priamo
 - Iba sprostredkované (napr. cez settery a gettery)
- Neviditeľné triedy
 - Nevieme vyrábať premenné typu neviditeľnej triedy
 - Nevieme konštruovať objekty neviditeľných tried
- Neviditeľné metódy
 - Nevieme ich zavolať
- Neviditeľné inštančné premenné
 - Nevieme robiť priamy prístup



Modifikátory viditeľnosti

- **Triedy** majú dva modifikátory viditeľnosti

- **public**

- Viditeľná všade

```
public class VerejnaTrieda {  
    ...  
}
```

- **(nič)**

- Viditeľná v svojom balíčku
- Neviditeľná v podbalíčkoch, nadbalíčkoch ani nikde inde

```
class BalíčkováTrieda {  
    ...  
}
```



Modifikátory viditeľnosti

- **Členovia triedy** majú štyri modifikátory viditeľnosti

- **public**

- Viditeľná všade

```
public int verejnaPremenná;
```

```
public void verejnaMetóda ();
```

- **(nič)**

- Viditeľná v svojom balíčku
 - Neviditeľná v podbalíčkoch, nadbalíčkoch ani nikde inde

```
int balíčkováPremenná;
```

```
void balíčkováMetóda ();
```



Modifikátory viditeľnosti

- **Členovia triedy** majú štyri modifikátory viditeľnosti
- **protected**
 - Viditeľná v svojom balíčku
 - Viditeľná aj v svojich potomkoch v iných balíčkoch

```
protected int chránenáPremenná;  
protected void chránenáMetóda();
```

- **private**
 - Viditeľná iba v svojej triede

```
private int súkromnáPremenná;  
private void súkromnáMetóda();
```



Modifikátory viditeľnosti

- **Členovia triedy** a ich viditeľnosť:

	trieda	package	podtrieda	inde
public	áno	áno	áno	áno
protected	áno	áno	áno	nie
(nič)	áno	áno	nie	nie
private	áno	nie	nie	nie



Modifikátory viditeľnosti

- Použitie závisí od konkrétneho návrhu
- V reálnych projektoch by mali byť modifikátory čo najprísnejšie
- Začínáme s **private** a iba keď máme **dobrý** dôvod nastavujeme voľnejšie modifikátory



Modifikátory viditeľnosti

- **public** by mali mať iba tie triedy a metódy, ktoré poskytneme iným programom a programátorom na používanie
- Typicky sú **public** metódy zo zadania, t.j. interfejsu
 - Interface má všetko public, aj keď to nepíšeme
- Inštančné premenné by nemali byť **nikdy public!**



Prehľad prednášky

- Abstraktné triedy
- Abstraktné metódy
- Interfejsy (rozhrania)
- Balíčky
- Modifikátory viditeľnosti
- **Triedenie**



Triedenie

- Triedenie je skoro v každom programe
 - Súbory podľa abecedy
 - Výrobky podľa ceny
 - Ceruzky podľa farby
- Tisíce spôsobov ako triediť
- Niektoré budú rozoberané v budúcom semestri
- Netreba zakaždým písať vlastné triedenie



Triedenie čísiel

- Triedenie čísiel v poli:

- `Arrays.sort(pole)`
 - je preťažená pre aj na všetky ostatné primitívy okrem **boolean**

```
int[] platy = new int[] {750, 340, 850, 400};  
  
Arrays.sort(platy);  
// pole je utriedené  
  
Arrays.toString(platy);
```

[340, 400, 750, 850]



Triedenie reťazcov

- Triedenie reťazcov
 - Triedime lexikograficky (ako v telefónnom zozname)
- Reťazec $a_1a_2a_3\dots a_n$ je v usporiadaní pred $b_1b_2b_3\dots b_n$
 - Ak buď $a_1 < b_1$ alebo
 - $\exists k \in [1, n]: \forall i < k$ platí $a_i = b_i$ a $a_k < b_k$
- Ak sú reťazce nerovnakej dĺžky a kratší je prefixom dlhšieho tak kratší je v usporiadaní pred dlhším



Triedenie reťazcov

- Triedenie reťazcov
 - "Pes" < "Veľryba", lebo P < V
 - "Pero" < "Pes", lebo "Pe" = "Pe" a r < s

```
String[] mená = new String[]{"Ján", "Jozef",  
"Alica", "Alexander"};
```

```
Arrays.sort(mená);  
// pole je utriedené
```

Alexander, Alica, Ján, Jozef



Triedenie objektov

- Čísla a reťazce mali prirodzené usporiadanie
- My však vieme triediť aj ľubovoľné objekty
- Musíme povedať čo to znamená, že jeden objekt je v usporiadaní pred druhým - to nie je vždy jasné
 - Matrix < Pacho, hybský zbojník
 - Lebo ich triedime podľa názvov
 - Lebo má horšie hodnotenie
 - Pacho, hybský zbojník < Matrix
 - Lebo má menej hercov
 - Lebo je kratší



Triedenie objektov

- Rozhodnutie vieme zaviesť do ľubovoľnej triedy implementovaním interfejsu (roly) `Comparable`
- Prekrývame metódu `compareTo()`

```
int compareTo(TypObjektu druhýObjekt)
```

- Máme vrátiť:
 - Menšie ako nula - ak objekt na ktorom sme volali `compareTo()` je v usporiadaní pred druhým objektom
 - Nula - ak sú v usporiadaní rovnaké
 - Väčšie ako nula - ak objekt na ktorom sme volali `compareTo()` je v usporiadaní za druhým objektom



Triedenie objektov

- Pre `Film` to vyzerá nasledovne:

```
public abstract class Film implements
Comparable<Film> {

    public int compareTo(Film inyFilm) {
        //vrátíme či náš nazovFilmu je pred
        //inyFilm.getNazovFilmu()
    }

}
```




Triedenie objektov

- Pre `Film` to vyzerá nasledovne:

```
public abstract class Film implements
Comparable<Film> {

    public int compareTo(Film inyFilm) {
        //vrátime či názovFilmu je pred
        //inyFilm.getNázovFilmu()
    }

}
```

Do `< >` uvádzame, akého typu budú objekty, s ktorými sa porovnávame. Použijeme našu triedu



Triedenie objektov

- Pre `Film` to vyzerá nasledovne:

```
public abstract class Film implements
Comparable<Film> {

    public int compareTo(Film inyFilm) {
        return nazovFilmu.compareTo(inyFilm
                                    .getNazovFilmu());
    }
}
```

Využijeme to, že `String`-y sa už vedia porovnávať podľa lexikografického usporiadania - implementujú rolu `Comparable<String>`



Triedenie objektov

- Triedime už bez problémov, tak ako pred tým:

```
Arrays.sort(zoznamFilmov);
```

- Čo však v prípade, že v jednom programe chcem riešiť triedenie aj podľa názvu aj podľa hodnotenia ?
 - Úplne bežná požiadavka
 - Neviem za behu meniť kód metódy `compareTo()`



Triedenie objektov

- Na porovnávanie dvoch objektov sa môžeme pozrieť z dvoch perspektív
 - Ja ako objekt sa porovnam s nejakým iným
 - Prídem ako nestranný pozorovateľ a porovnam dva objekty a poviem, ktorý bude pred ktorým
- Prvá perspektíva bola použitá pri metóde `compareTo()`
 - default zotriedenie
- Druhú perspektívu vyriešime vytvorením novej triedy, ktorá implementuje interfejs `Comparator` s jedinou metódou:

```
int compare(TypObjektu prvýObjekt, TypObjektu druhýObjekt)
```



Triedenie objektov

```
public class FilmPodlaMenaComparator implements
Comparator<Film> {

    public int compare(Film film1, Film film2) {
        return film1.getNazovFilmu().compareTo(film2
.getNazovFilmu());
    }
}
```

```
public class FilmPodlaDlzkkyComparator implements
Comparator<Film> {

    public int compare(Film film1, Film film2) {
        return film1.getDlzkkaFilmu() - film2
.getDlzkkaFilmu());
    }
}
```



Triedenie objektov

- Triedime podľa čoho chceme

```
Arrays.sort(zoznamFilmov, new FilmPodlaMenaComparator());  
// pole je utriedené podľa mena
```

```
Arrays.sort(zoznamFilmov, new FilmPodlaDlzkkyComparator());  
// pole je utriedené podľa dĺžky filmu
```



Triedenie v opačnom poradí

- Chceme triediť od najlepších hodnotení
- Nemusíme robiť nový komparátor, stačí hotový obrátiť

```
Comparator<Film> porovnavac = new  
    FilmPodlaDlzklyComparator();  
  
Arrays.sort(zoznamFilmov,  
    Collections.reverseOrder(porovnavac));  
// pole je utriedené podľa dĺžky filmov zostupne
```



Chceme triediť po slovensky

- Chceme triediť tak, ako nás učia jazykovedci

```
String[] mená = new String[]{"Adam", "Cecília",  
"Cháron", "Ábel", "Daniel"};
```

```
Arrays.sort(mená);
```

```
// pole je utriedené, ale nejako nedobre
```

Adam, Cecília, Cháron, Daniel, Ábel

- Na vine je lexikografické usporiadanie
 - Diakritické znaky sú za A-Z
 - Ce < Ch, lebo C = C a e < h



Chceme triediť po slovensky

- Milí autori Javy za nás vyrobili komparátor pre slovenčinu
 - `java.util.Collator`

```
String[] mená = new String[]{"Adam", "Cecília",  
"Cháron", "Ábel", "Daniel"};
```

```
Collator skPorovnavac =  
    Collator.getInstance(new Locale("sk"));
```

```
Arrays.sort(mená, skPorovnavac);
```

Adam, Ábel, Cecília, Daniel, Cháron



Ďakujem za pozornosť !

