11. prednáška (27.11.2017)

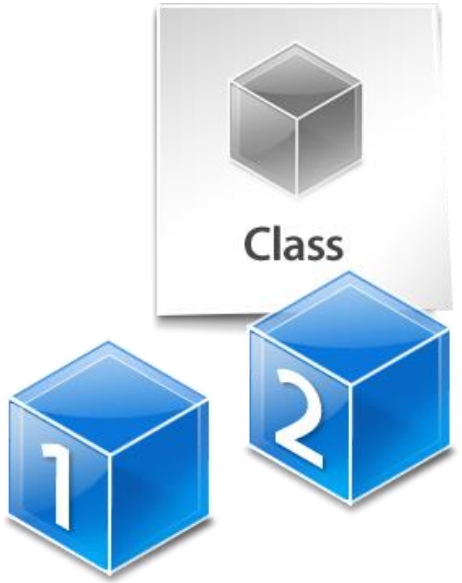
Modifikátory, rozhrania a všeličo iné...

Kľúčové koncepty OOP

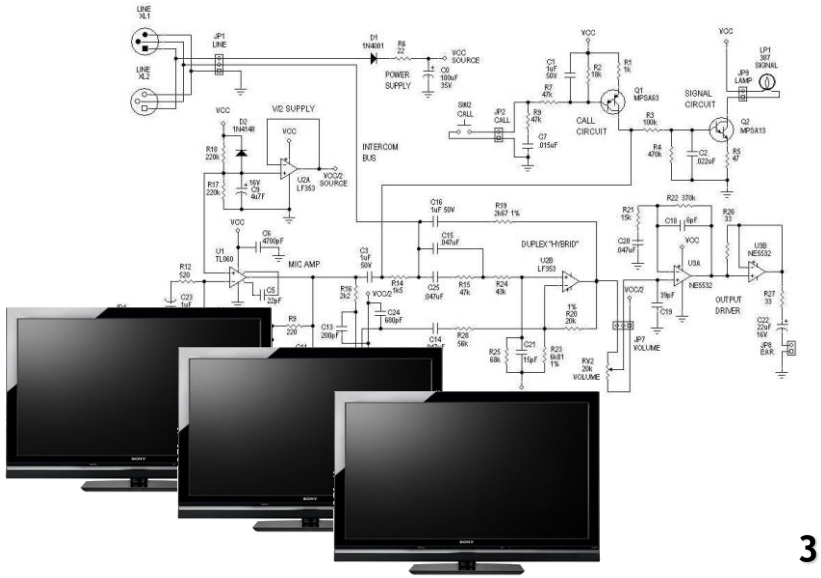
- Čo je **trieda**? Čo je **objekt**? Aký je vzťah medzi objektom a triedou?
- Referencia na objekt, premenné referenčného typu
- Vytváranie **nových tried rozširovaním** existujúcich
  - Trieda `Object`
  - **Dedičnosť** (inheritance)
  - **Prekrývanie metód** (override)
- Vytváranie objektov (inštancií) tried
  - **Konštruktor**(y)
- **Zapúzdrenie** (encapsulation)
- **Polymorfizmus**

Čo je to trieda?

- **Trieda** je **šablóna** (vzor), ktorý **predpisuje** aké **inštančné premenné** a aké **metódy** majú objekty danej triedy a čo sa udeje pri zavolaní týchto metód

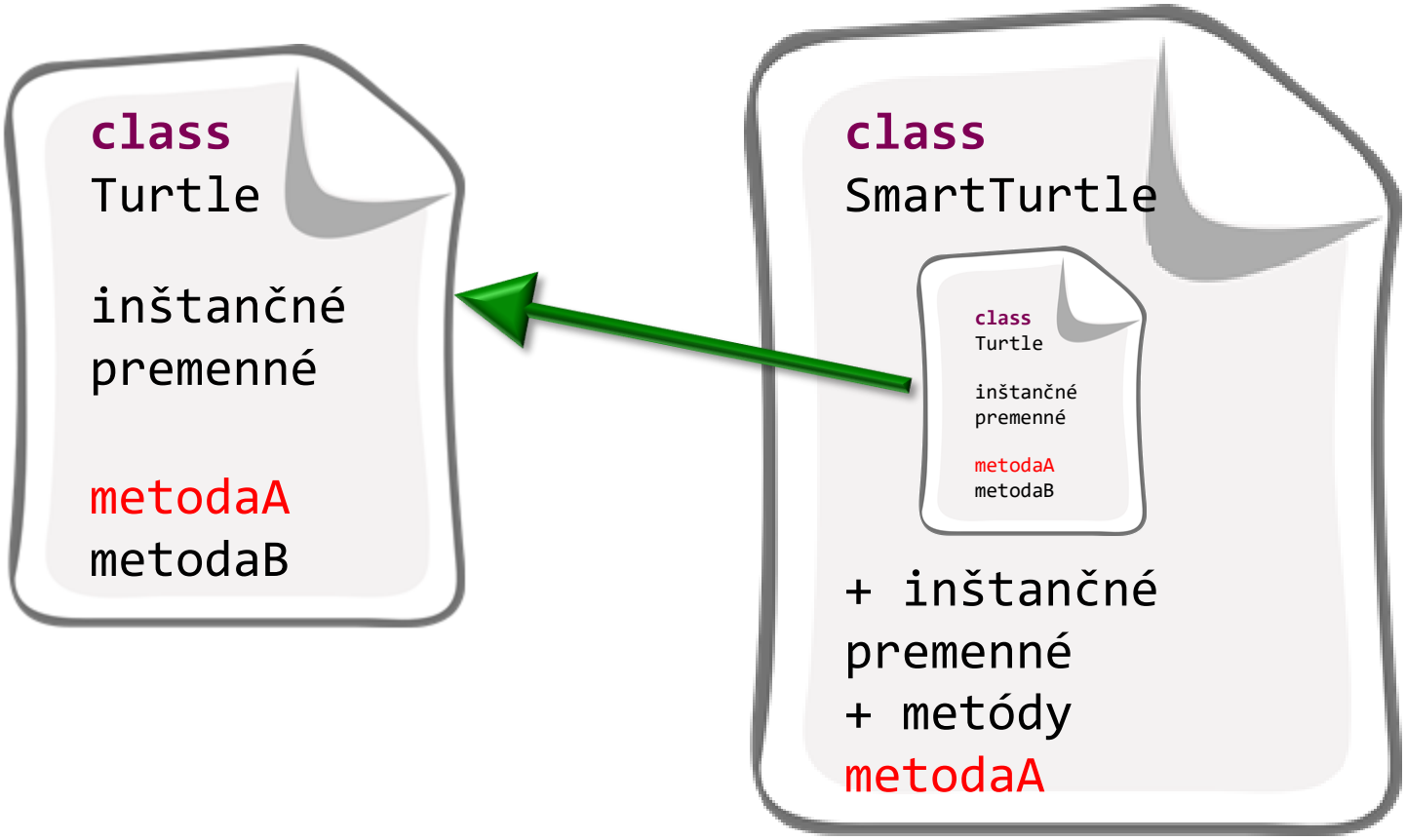



```
class
Turtle

inštančné
premenné


metódy
```



Rozširovanie a prekrývanie

```
public class SmartTurtle extends Turtle
```



Konštruktory

- Každá trieda má **aspoň jeden konštruktor**
  - konštruktory sa nededia (ale konštruktor rodiča sa dá zavolať)
  - ak programátor túto podmienku nesplní, vytvára sa implicitný (bezparametrový) konštruktor volajúci bezparametrový konštruktor rodičovskej triedy
- **Prvý príkaz konštruktora** musí byť volanie konštruktora rodičovskej triedy (`super(…)`) alebo iného konštruktora vytváranej triedy (`this(…)`)
  - ak toto nie je splnené, Java dopĺňa `super(…)`
  - konštruktor (z rodiča alebo iný z triedy) sa môže volať len ako prvý príkaz konštruktora

Premenné referenčného typu

```
Turtle franklin;
```

- Premenná `franklin` môže referencovať len objekty triedy `Turtle` **a tried, ktoré rozširujú triedu `Turtle`**

```
franklin.metoda()
```

- Cez premennú `franklin` môžeme volať len metódy **definované** v triede `Turtle`
- Polymorfizmus: Nevieme, aká implementácia volanej metódy sa vykoná, keďže trieda aktuálne referencovaného objektu mohla volanú metódu prekryť svojou implementáciou

Pretypovanie referencií

- `referencia instanceof Trieda`
  - má trieda aktuálne referencovaného objektu niekde medzi svojimi predkami triedu `Trieda` alebo ide o triedu `Trieda`?
- Referenciu ide explicitne pretypovať (programátor preberá zodpovednosť)
  - `Trieda o = (Trieda)referencia;`
  - `((Trieda)referencia).metoda(…);`

Dnes...

Zoznam/Správca filmov

```
public class ZoznamFilmov
```

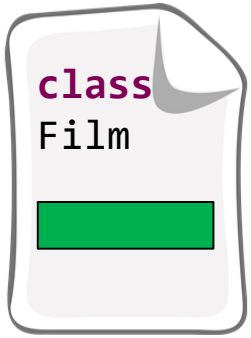
```
public class Film
```

```
public class FilmNaPaske
```

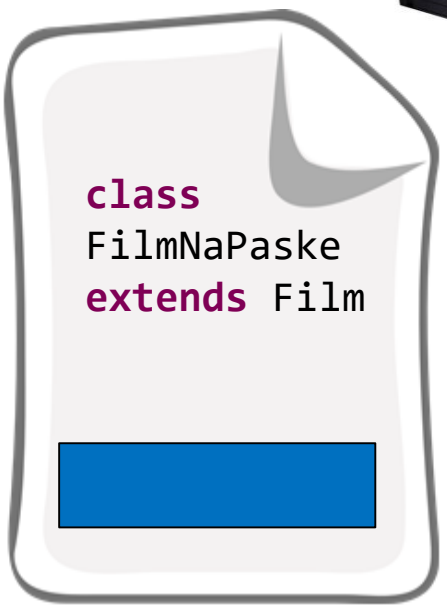
```
public class FilmNaDvd
```

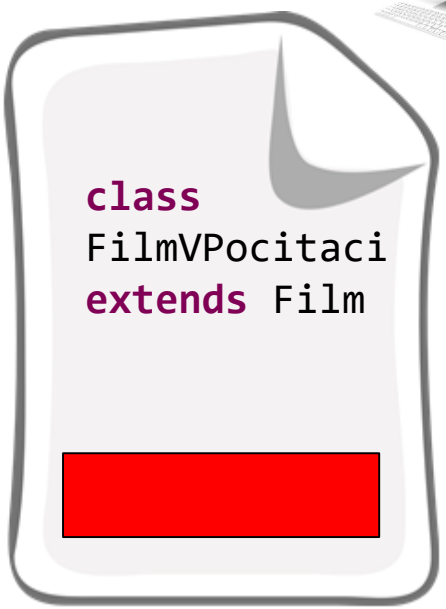
```
public class FilmVPocitaci
```

Návrh tried

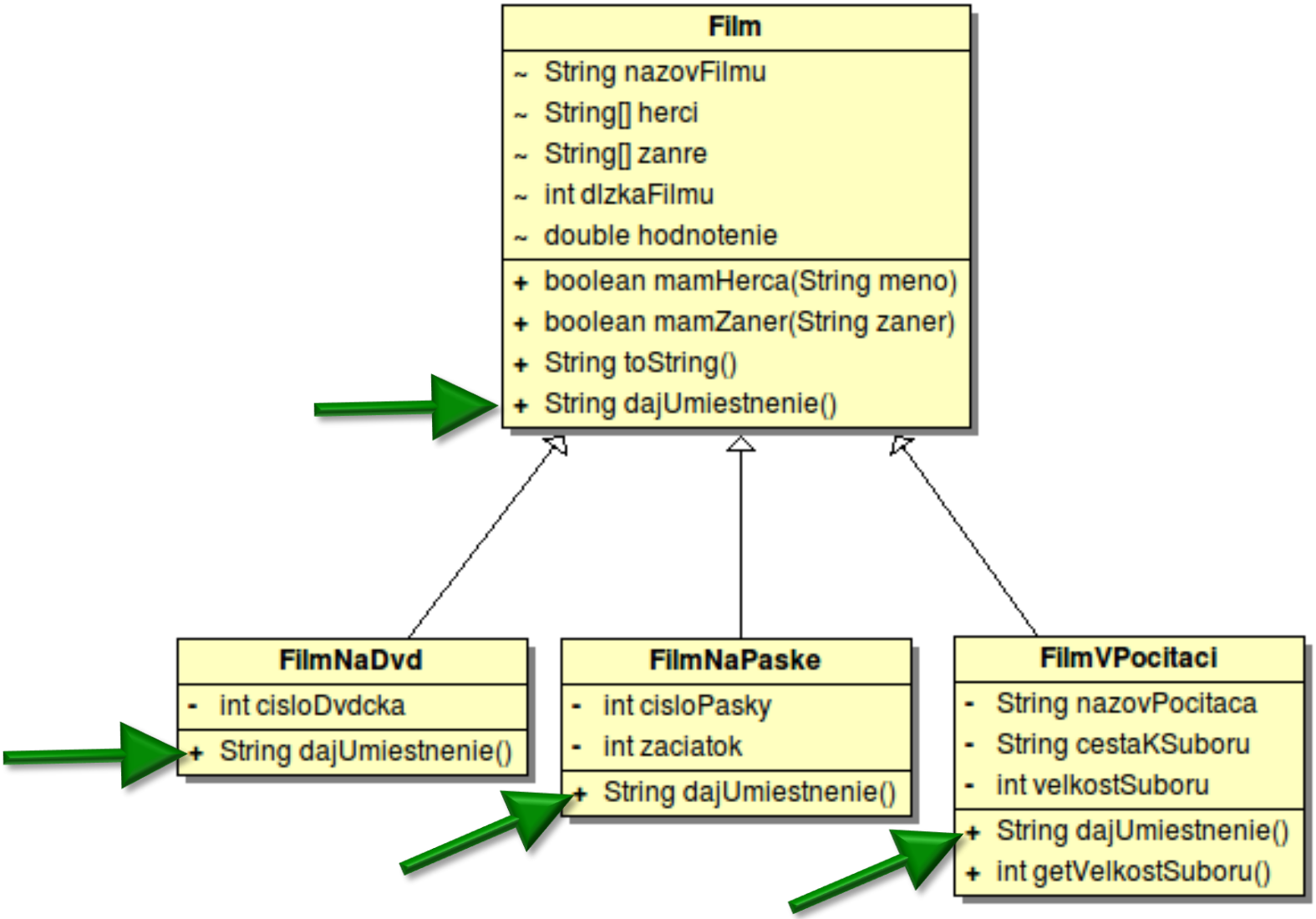

class
Film


class
FilmNaDvd
extends Film


class
FilmNaPaske
extends Film


class
FilmVPocitaci
extends Film

Návrh tried


Film
~ String nazovFilmu
~ String[] herci
~ String[] zanre
~ int dlzkaFilmu
~ double hodnotenie
+ boolean mamHerca(String meno)
+ boolean mamZaner(String zaner)
+ String toString()
+ String dajUmiestnenie()
FilmNaDvd
- int cisloDvdcka
+ String dajUmiestnenie()
FilmNaPaske
- int cisloPasky
- int zaciatok
+ String dajUmiestnenie()
FilmVPocitaci
- String nazovPocitaca
- String cestaKSuboru
- int velkostSuboru
+ String dajUmiestnenie()
+ int getVelkostSuboru()

Modifikátory

- Rôzne „magické“ slovíčka, ktoré upravujú isté vlastností tried, metód, premenných, ...

```
class
Turtle

inštančné
premenné

metódy
```

Problém

- Trieda `Film`
  - obsahuje spoločné inštančné premenné a metódy pre triedy `FilmNaPaske`, `FilmNaDvd`, `FilmVPocitaci`
  - neobsahuje žiadne umiestnenie
- V reálnom programe nikto rozumný nespraví `new Film(…)`, lebo to v kontexte celého projektu nedáva zmysel
  - ale aj taký sa skôr či neskôr nájdu...

Abstraktné triedy

```
public abstract class Film {
    ...
}
```

Modifikátor triedy

- Abstraktná trieda =
  - označená modifikátorom `abstract`
  - zákaz vytvárania inštancií tejto triedy cez **`new`**

Problém

- Metóda `dajUmiestnenie` v triede `Film`
  - **potrebujeme** ju, aby sme mali istotu, že každý film vie „povedať“ svoje umiestnenie
  - **očakávame**, že ju tvorcovia rozširujúcich tried rozumne prekryjú
  - priamo v tiede Film jej **nevieme dať rozumnú implementáciu**

- Čo ak tvorca rozširujúcej triedy zabudne metódu `dajUmiestnenie` prekryť?
  - aj taký sa skôr či neskôr nájde...

Abstraktné metódy

- Abstraktné metódy =
  - sú označené modifikátorom `abstract`
  - žiadne telo (implementácia)
  - dedia sa (ako všetky metódy)
  - môžu sa vyskytovať len v abstraktnej triede

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

žiadne { }

Modifikátor metódy

Abstraktné metódy a triedy

- Trieda má aspoň jednu abstraktnú metódu (vlastnú alebo zdedenú):
  - (sedliacky rozum) Ak trieda obsahuje **aspoň jednu abstraktnú metódu**, musí byť **abstraktná** (=zákaz vytvorenia inštancie)

- Dôsledok: Potomkovia triedy musia byť abstraktní aspoň do chvíle, kým **prekrytím neposkytnú implementáciu** všetkým zdedeným abstraktným metódam.

Abstraktné metódy a triedy

- Abstraktná trieda a abstraktná metóda v nej nám zabezpečia, že v poli filmov sú iba objekty takých tried, ktoré majú prekrytú 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());
   }
 }
...
}
```

Modifikátor final

- Keď volám svoju metódu, nemám istotu, že mi ju niekto v rámci rozširovania neprekryl...
  - zvyčajne to chceme dovoliť, ale nie vždy sa to hodí
- Modifikátor **`final`**
  - `final` trieda = zákaz rozširovania
  - `final` metóda = zákaz prekrývania
  - `final` inštančná premenná = hodnotu môžem priradiť (nastaviť) len raz a to v konštruktore, ... neskôr sa nedá meniť
  - `final` lokálna premenná = hodnotu môžem priradiť len raz, ... neskôr sa nedá meniť

Modifikátory viditeľnosti

- Pomocou **modifikátorov viditeľnosti** vieme nastaviť **viditeľnosť** tried, metód a inštančných premenných
- S tým, čo **nevidíme, nevieme pracovať** priamo
  - iba sprostredkovane (napr. cez settery a gettery)

- 4 typy (nie všade ide použiť každý jeden):
  - `public`
  - `protected`
  - (nič) – defaultný, resp. package-private
  - `private`

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

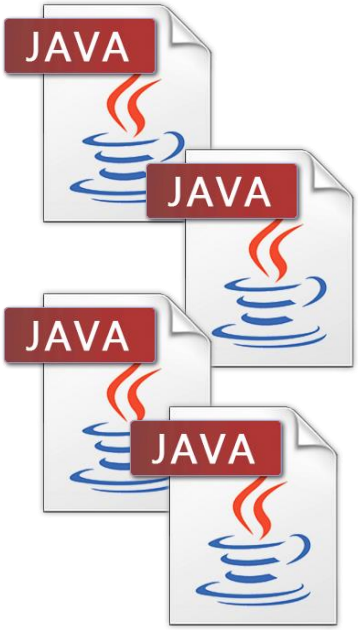
Modifikátory viditeľnosti

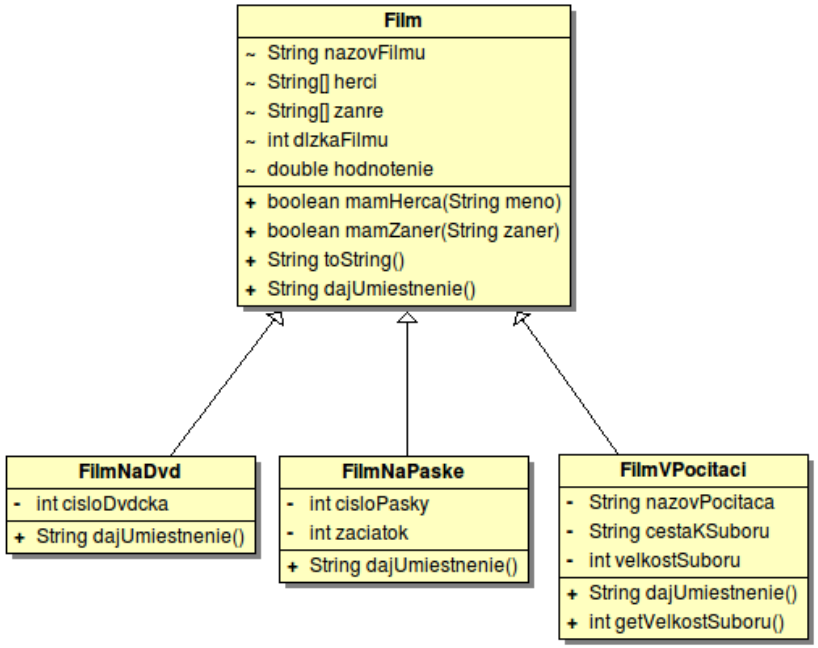
- Použitie závisí od konkrétneho návrhu
- V reálnych projektoch by mali byť modifikátory čo najprísnejšie
- Začíname s `private` a iba keď máme dobrý dôvod nastavujeme voľnejšie modifikátory
- `public` by mali mať iba tie triedy a metódy, ktoré poskytneme iným programom a programátorom na používanie
- Inštančné premenné by nemali byť nikdy `public`!

Defaultný balíček

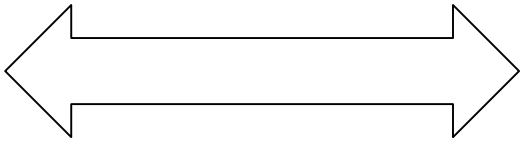
- Defaultný balíček = balíček bez mena
- Triedy v defaultnom balíčku **nemožno importovať** a **nemožno použiť** v triedach z iných balíčkov
  - nevytvárame triedy v defaultnom balíčku; výnimkou môžu byť nejaké drobné experimentálne minikódy...

Rozhrania





Program, zdrojový kód



Reálny svet

A čo tak správa hudby?

- Pridajme správu hudby...
- Pieseň (Song):
  - Názov
  - Interpret
  - Dĺžka v sekundách
  - Umiestnenie (kde ju hľadať)
- Možné rozšírenia podľa umiestnenia:
  - Pieseň na platni/CD-čku
  - Pieseň na páske
  - Pieseň v počítači

Playlist

- Playlist = usporiadaný zoznam vecí na prehranie...
- Môže obsahovať:
  - piesne?
  - filmy?
  - zoznamy filmov?
  - zoznamy piesní?

Playlist

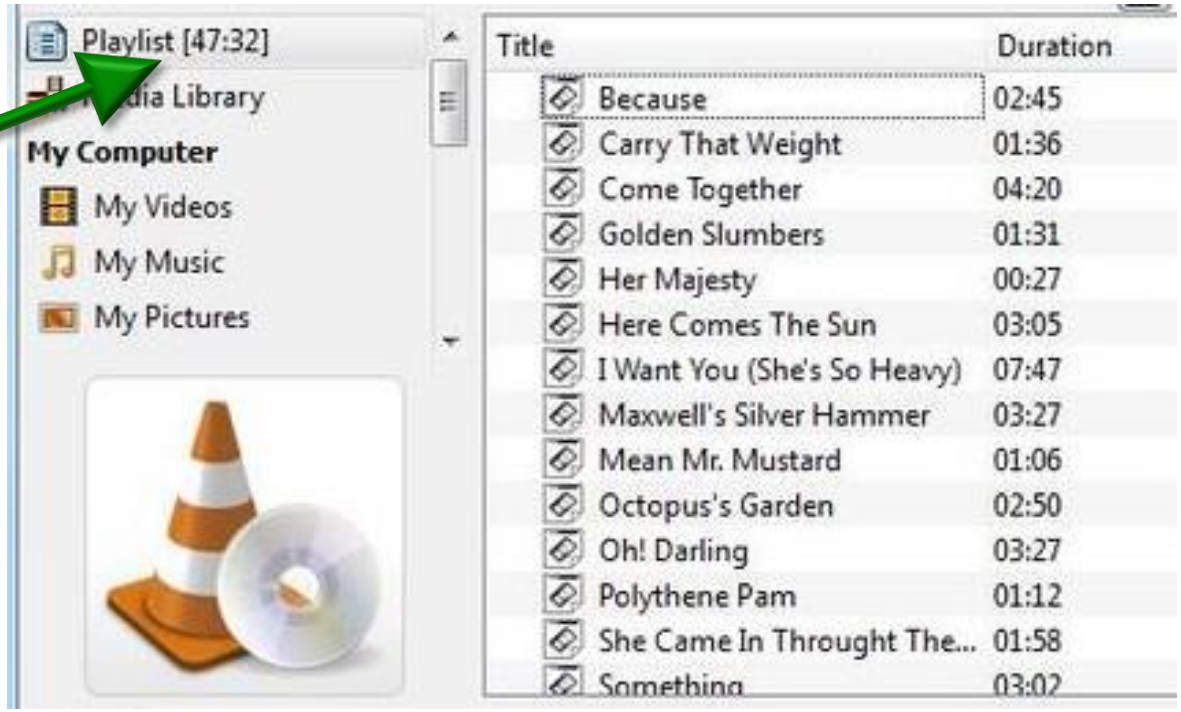
```
public class Playlist {
    private ???[] polozky;
}
```

- Akú funkcionalitu očakávame od playlistu?
- Zoznam čoho je playlist?
  - Čo iné by ešte mohlo byť v playliste?
- Čo očakávame od položky v playliste?

Položka v playliste

- Od položky v playliste očakávame:
  - vie povedať, aké ma trvanie (duration)
  - má nejaký názov/popis (title)

Playlist by mal vedieť vypočítať celkové trvanie.

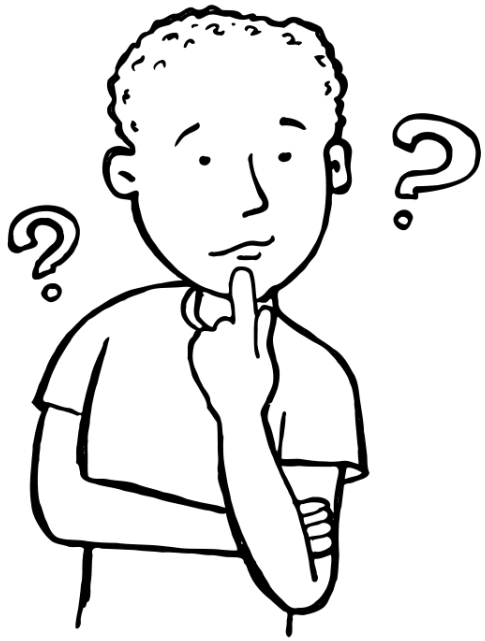


PolozkaVPlayliste

```
public class PolozkaVPlayliste {

  public int getTrvanieVSekundach() {
    …
  }

  public String getNazov() {
    …
  }

}
```

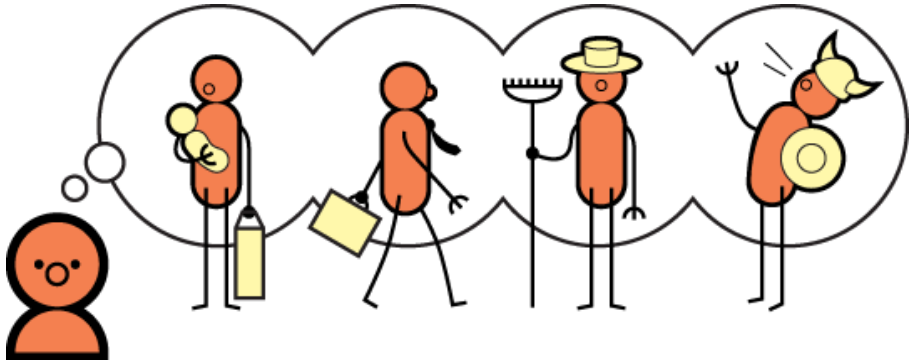
PolozkaVPlayliste

- `FilmNaDvd` je `Film`?
- `Film` je `PolozkaVPlayliste`?
- `Film` môže vystupovať ako `PolozkaVPlayliste`?
- `Pesnicka` je `PolozkaVPlayliste`?
- `Pesnicka` môže vystupovať ako `PolozkaVPlayliste`?
- `ZoznamFilmov` je `PolozkaVPlayliste`?
- `ZoznamFilmov` môže vystupovať ako `PolozkaVPlayliste`?



Trieda vs. rola

- Objekt je inštanciou jednej triedy.
- Trieda rozširuje práve jednu inú triedu.
- Trieda popisuje:
  - **čo** (aké metódy) a
  - **ako** (implementácia metód, inštančné premenné, konštruktory).
- Rola/kontrakt hovorí:
  - **čo** (aké metódy)



Rozhranie (interface)

- Rola v Jave = rozhranie

```
public interface PolozkaVPlayliste {
  public int getTrvanieVSekundach();
  public String getNazov();
}
```

- **Rozhranie** = zoznam hlavičiek metód
  - žiadna implementácia
  - žiadne inštančné premenné
  - žiadne konštruktory
  - len hlavičky public metód

Rozhranie vs. trieda

```
public class Film extends Object implements PolozkaVPlayliste {
  …
}
```

- Trieda rozširuje len jednu triedu, ale môže **implementovať veľa rozhraní**

```
... implements Rozhranie1, Rozhranie2 {…
```

- Ak trieda implementuje rozhranie, musí **mať všetky metódy**, ktoré sú uvedené v tomto rozhraní

Premenné referenčného typu

```
Rozhranie objekt;
```

- Premenná `objekt` môže referencovať objekt ľubovoľnej triedy, ktorá cez **`implements`** prehlásila, že implementuje rozhranie `Rozhranie`

```
objekt.metoda()
```

- Cez premennú `objekt` môžeme volať **len metódy** definované v rozhraní `Rozhranie`.
- To, aká implementácia sa vykoná, záleží len od triedy referencovaného objektu.

Rozširovanie rozhraní

```
public interface RozsireneRozhranie
          extends Rozhranie1, Rozhranie2 {

   …

}
```

- `RozsireneRozhranie` bude obsahovať:
  - všetky hlavičky metód z rozhrania `Rozhranie1`
  - všetky hlavičky metód z rozhrania `Rozhranie2`
  - všetky hlavičky metód, ktoré sme explicitne napísali do rozhrania `RozsireneRozhranie`

Sumarizácia rozhraní

- Interface = pomenovaný **zoznam hlavičiek** metód
  - hlavička metódy = názov, návratový typ, zoznam typov parametrov

```
public interface Rozhranie { ... }

public class Trieda implements Rozhranie { ... }
```

Trieda prehlasuje, že bude mať všetky metódy, ktoré sú uvedené v rozhraní.

```
Rozhranie o = …;
```

Premenná o je schopná referencovať objekt ľubovoľnej triedy, ktorá prehlásila, že implementuje interface Rozhranie

Playlist

```
public class Playlist {

  private PolozkaVPlayliste[] polozky;

  public Playlist() {
    polozky = new PolozkaVPlayliste[0];
  }

  public void pridaj(PolozkaVPlayliste polozka) {
    …
  }

  ...
}
```

Usporiadavanie

- Usporiadavanie (triedenie) je skoro v každom programe
  - súbory podľa abecedy
  - výrobky podľa ceny
  - ...
- Preskúmaný problém, kopy rôznych riešení
  - viac na PAZ1b
- Netreba zakaždým písať vlastnú implementáciu

Usporiadanie čísiel

- Usporiadanie čísiel v poli:
  - `Arrays.sort(pole)`
    - je preťažená pre aj na všetky ostatné primitívne typy okrem **`boolean`**

```
int[] platy = new int[]{750,340,850,400};

Arrays.sort(platy);
// pole je utriedené

Arrays.toString(platy);
```

☐[340, 400, 750, 850]

Usporiadanie reťazcov

- Usporiadanie reťazcov
  - lexikograficky (ako v telefónnom zozname)
- Reťazec $a_1a_2a_3...a_n$ je v usporiadaní pred $b_1b_2b_3...b_n$
  - 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 nemajú reťazce rovnakú dĺžku, kratší má akoby koncové znaky doplnené znakom s kódom -1

Usporiadanie reťazcov

- Usporiadanie 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

Usporiadanie po slovensky

- Chceme usporiadať tak, ako nás učia jazykovedci

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

Arrays.sort(mená);
// pole je usporiadané, 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

Usporiadanie objektov

- Čísla a reťazce mali prirodzené usporiadanie
- Ako usporiadať ľubovoľné objekty?
  - musíme nejako 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

Usporiadanie objektov

- Rozhodnutie vieme zaviesť do ľubovoľnej triedy implementovaním rozhrania (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 (je menší)
  - Nula – ak sú v usporiadaní rovnaké
  - Väčšie ako nula - ak objekt na ktorom sme volali `compareTo()` je v usporiadaní za druhým objektom (je väčší)

Usporiadanie objektov

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

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

- Máme vrátiť:
  - `a.compareTo(b) < 0` ak „a < b”
  - `a.compareTo(b) == 0` ak „a == b”
  - `a.compareTo(b) > 0` ak „a > b”

Usporiadanie objektov

- Pre `Film` to vyzerá nasledovne:

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

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

}
```

Usporiadanie objektov

- Pre `Film` to vyzerá nasledovne:

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

  public int compareTo(Film inyFilm) {
    //vrátime či n  nazovFilmu je pred
    //inyFilm.getNaz  ilmu()
  }

}
```

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

Usporiadanie 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>`

Usporiadanie objektov

- Usporiadavame už bez problémov:

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

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

Usporiadanie objektov

- Na porovnávanie dvoch objektov sa môžem pozrieť z dvoch perspektív
  1. Ja, ako objekt, sa porovnám s nejakým iným
  2. Prídem ako nestranný pozorovateľ, porovnám 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 rozhranie `Comparator` s jedinou metódou:

```
int compare(TypObjektu o1,TypObjektu o2)
```

Comparator<Trieda>

- Rozhranie `Comparator<TypObjektu>`:

```
int compare(TypObjektu a,TypObjektu b)
```

- Máme vrátiť:
  - `compare(a, b) < 0` ak „a < b”
  - `compare(a, b) == 0` ak „a == b”
  - `compare(a, b) > 0` ak „a > b”
- Užitočné metódy:
  - `Integer.compare(a, b)`
  - `Double.compare(a, b)`
  - …

Usporiadanie objektov

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

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

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

   public int compare(Film film1,Film film2) {
      return Integer.compare(film1.getDlzkaFilmu(),
                             film2.getDlzkaFilmu());
   }
}
```

Usporiadanie objektov

- Usporiadavame podľa čoho chceme:

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

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

Opačné usporiadanie

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

```
Comparator<Film> porovnavac = new
                              FilmPodlaDlzkyComparator();

Arrays.sort(zoznamFilmov,
                    Collections.reverseOrder(porovnavac));
// pole je utriedené podľa dĺžky filmov zostupne
```

Usporiadanie po slovensky

- `java.text.Collator` – Comparator, ktorý vie usporiadať reťazce po slovensky:

```
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 !