12. prednáška (13.12.2021)

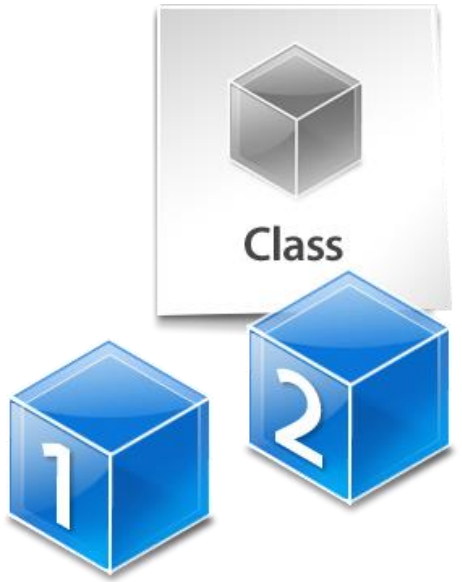
Modifikátory, rozhrania a všeličo iné...

Kľúčové koncepty OOP

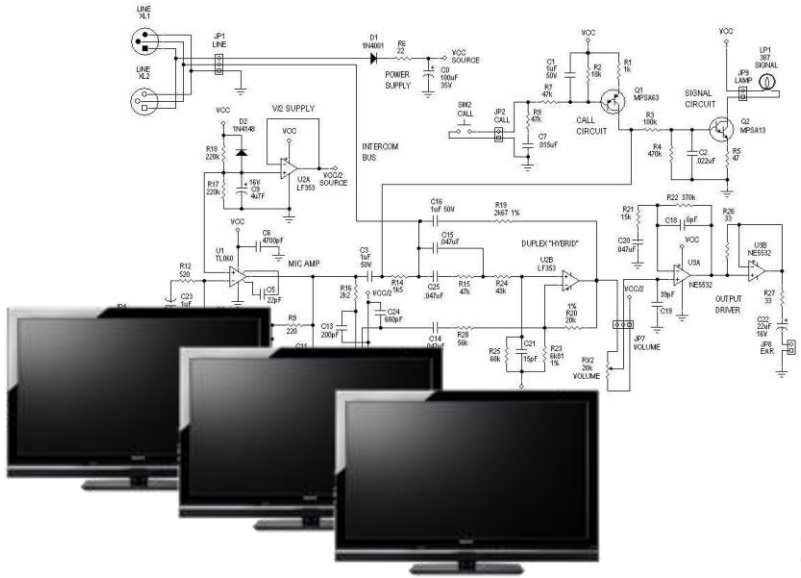
- Čo je **trieda**? Čo je **objekt**? Aký je vzťah medzi objektom a triedou?
- Referencia na objekt, premenné referenčného typu
- Vytváranie **nových tried rozširovaním** existujúcich
  - Trieda `Object`
  - **Dedičnosť** (inheritance)
  - **Prekrývanie metód** (override)
- Vytváranie objektov (inštancií)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

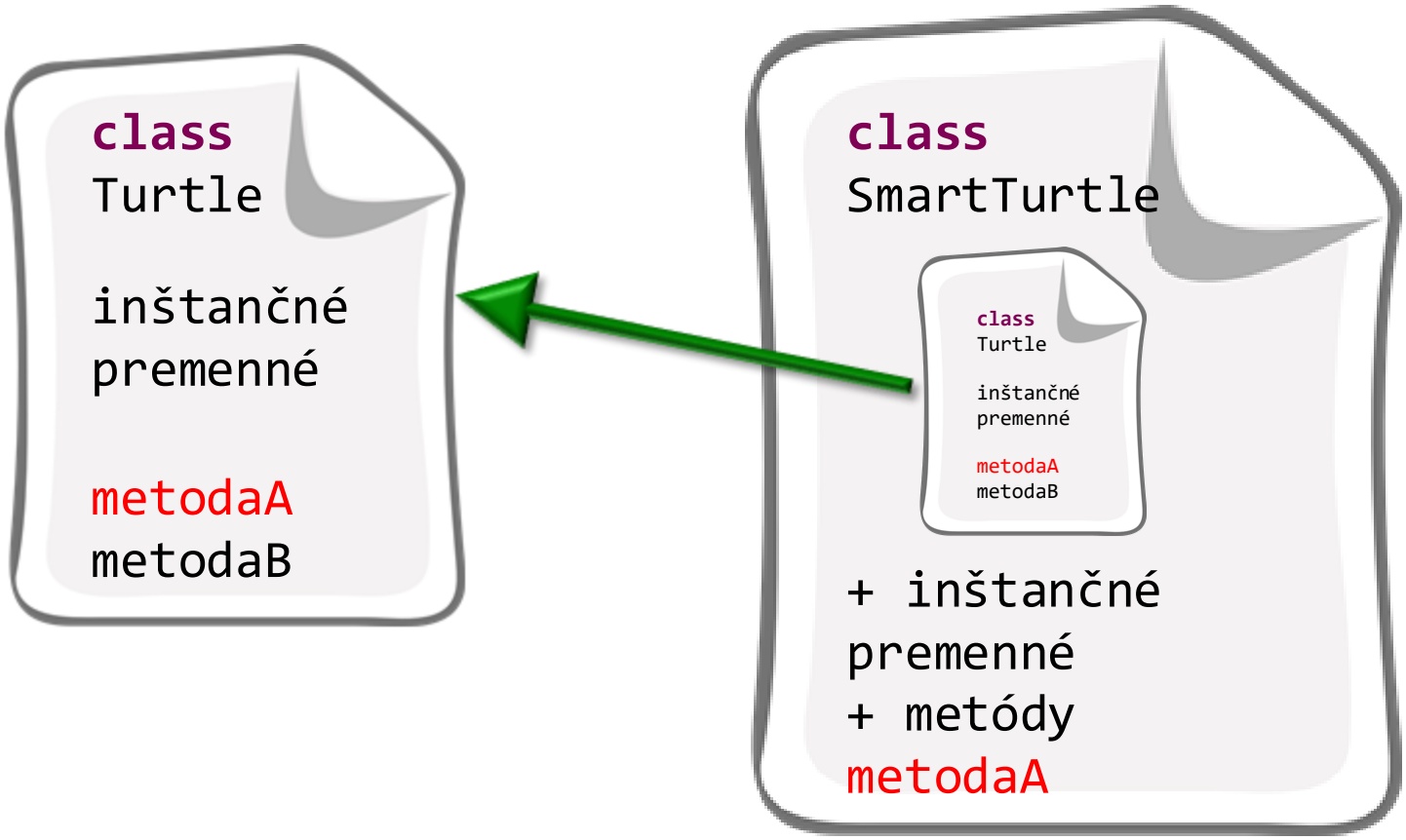



```
class
Turtle

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

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

Rozhrania, ktoré sme už videli

- java.util.**Scanner**, java.util.**PrintWriter**
  - Closeable/AutoCloseable
- java.util.**ArrayList**<E>, java.util.**LinkedList**<E>
  - **List**<E>, Iterable<E>, Cloneable, Collection<E>
- java.util.**TreeSet**<E>
  - Set<E>, SortedSet<E>, Collection<E>

A čo tak správa hudby?

- Pridajme správu hudby...
- Pieseň (Song):
  - Názov
  - Interprét
  - Dĺžka v sekundách
  - Umiestnenie (kde ju hľadať)
- Možné rozšírenia podľa umiestnenia:
  - Pieseň na Spotify
  - Pieseň na platni

Playlist

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

Playlist

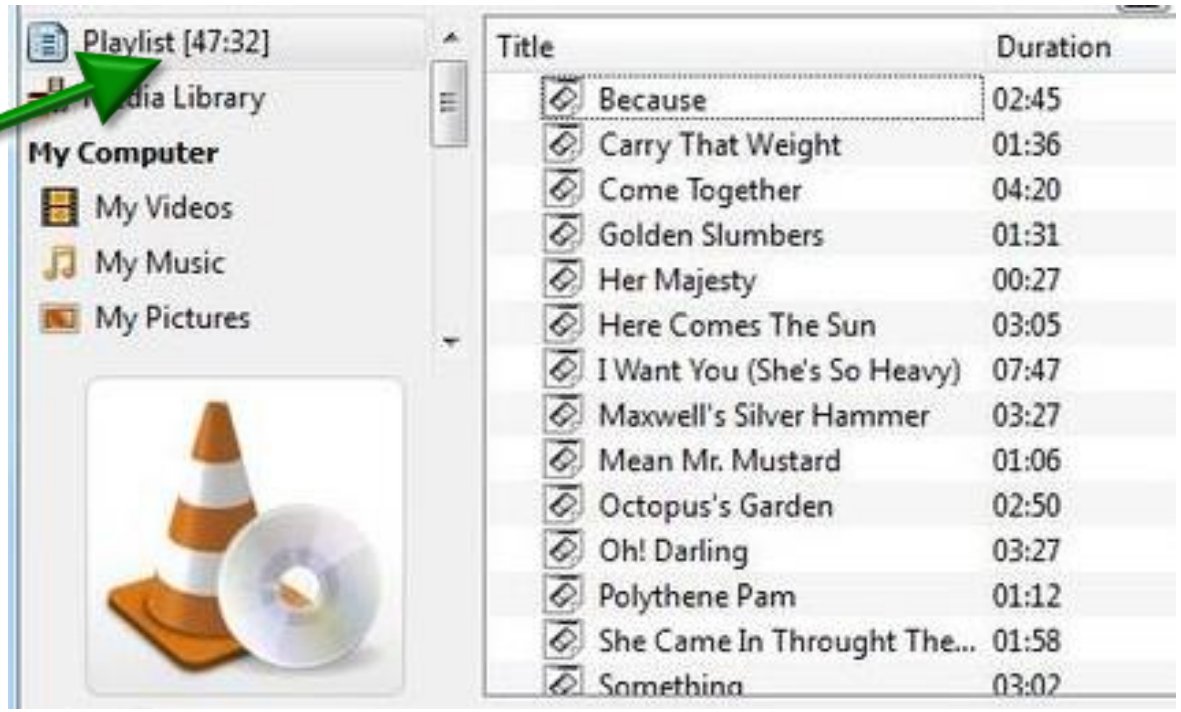
```
public class Playlist {
    private ???[] polozky;
}
```

- 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.

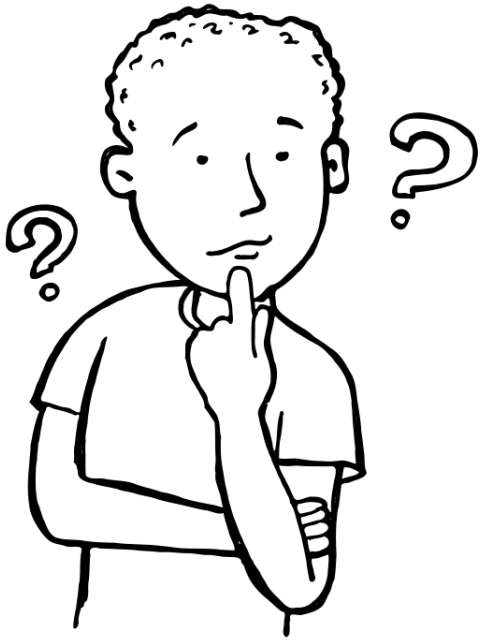


Položka v playliste

```
public class PlaylistItem {

  public int getDuration() {
    …
  }

  public String getTitle() {
    …
  }

}
```

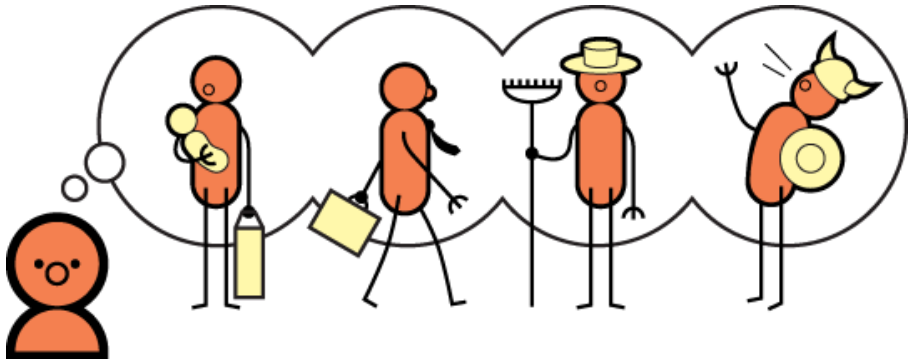
Playlist Item

- `AudioBook` je `PlaylistItem`?
- `AudioBook` môže vystupovať ako `PlaylistItem`?
- `Song` je `PlaylistItem`?
- `Song` môže vystupovať ako `PlaylistItem`?
- `Library` je `PlaylistItem`?
- `Library` môže vystupovať ako `PlaylistItem`?



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 PlaylistItem{
  int getDuration();
  String getTitle();
}
```

```
public interface PlaylistItem{
  public int getDuration();
  public String getTitle();
}
```

- **Rozhranie** ≈ zoznam hlavičiek metód
  - žiadna implementácia
  - žiadne inštančné premenné
  - žiadne konštruktory
  - len hlavičky **public** metód (public nemusíme písať)

default metódy, statické premenné

Rozhranie vs. trieda

```
public class AudioBook extends Book implements PlaylistItem {
  …
}
```

- 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`
- *pre fajnšmekrov* - default implementácie metód

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 PlaylistItem[] items;

  public Playlist() {
    items = new PlaylistItem[0];
  }

  public void pridaj(PlaylistItem item) {
    …
  }

  ...
}
```

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:

[340, 400, 750, 850]

  - `Arrays.sort(pole)`
    - je preťažená na všetky triedy rozširujúce *Object* aj na všetky ostatné primitívne typy okrem **`boolean`**
    - návratový typ **`void`**
    - verzia pre prvky v *List*-e: **`Collections.sort()`**

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

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

Arrays.toString(platy);
```

Usporiadanie reťazcov

- Usporiadanie reťazcov
  - lexikograficky (podľa abecedy)
- 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[]{”Peter", ”Tomáš", ”Šimon",
”Dávid”, ”Chuck”};

Arrays.sort(mená);
// pole je usporiadané, ale nejako nedobre
```

Chuck, Dávid, Peter, Tomáš, Šimon

- 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:
    - Harry Potter 4 < Na západe nič nové
      - Lebo ich triedime podľa názvov
      - Lebo má horšie hodnotenie
    - Harry Potter 4 > Na západe nič nové
      - Lebo ich triedime podľa autorov
      - Lebo má viac strán

Usporiadanie objektov

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

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

- **this** vs. **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 `Movie` to vyzerá nasledovne:

```
public class Book implements Comparable<Book>{

  public int compareTo(Book anotherBook) {
    //vrátime či náš title je pred
    //anotherBook.getTitle()
  }

}
```

Usporiadanie objektov

- Pre `Movie` to vyzerá nasledovne:

```
public class Book implements Comparable<Book>{

   public int compareTo(Book anotherBook) {
      //vrátime či náš title je pred
      //anotherBook.getTitle()
   }


}
```

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

Usporiadanie objektov

- Pre `Movie` to vyzerá nasledovne:

```
public class Book implements Comparable<Book>{


   public int compareTo(Book anotherBook) {
      return title.compareTo(anotherBook
                             .getTitle());
   }
}
```

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

Usporiadanie objektov

- Usporiadavame už bez problémov:

```
Arrays.sort(zoznamKnih);
```

- 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)
```

- o1 vs. 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 BookByTitleComparator implements
Comparator<Book> {

  public int compare(Book book1, Book book2) {
    return book1.getTitle().compareTo(book2.getTitle());
  }
}
```

```
public class BookByRatingComparator implements
Comparator<Book> {

  public int compare(Book book1, Book book2) {
      return Double.compare(book1.getRating(),
                    book2.getRating());
  }
}
```

Usporiadanie objektov

- Usporiadavame podľa čoho chceme:

```
Arrays.sort(zoznamKnih, new BookByTitleComparator());
// pole je utriedené podľa mena

Arrays.sort(zoznamKnih, new BookByRatingComparator());
// pole je utriedené podľa hodnotenia
```

Opačné usporiadanie

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

```
Comparator<Book> porovnavac = new
                              BookByRatingComparator();

Arrays.sort(zoznamKnih,
                    Collections.reverseOrder(porovnavac));
// pole je utriedené podľa hodnotenia zostupne
```

Usporiadanie po slovensky

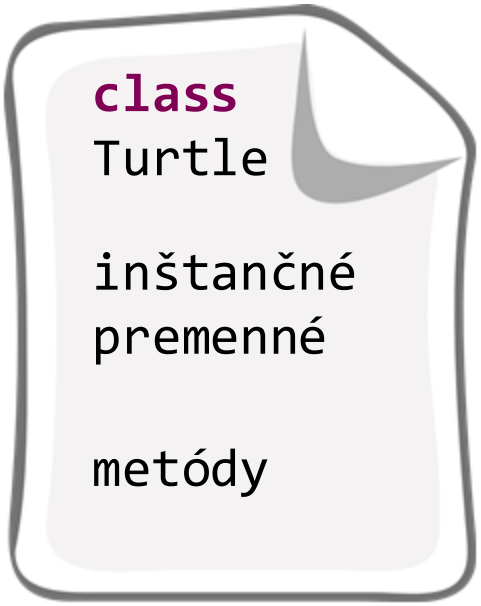
- `java.text.Collator` – Comparator, ktorý vie usporiadať reťazce po slovensky:

```
String[] mená = new String[]{”Peter", ”Tomáš", ”Šimon",
”Dávid”, ”Chuck”};

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

Arrays.sort(mená, skPorovnavac);
```

Dávid, Chuck, Peter, Šimon, Tomáš

Modifikátory

- Rôzne „magické“ slovíčka, ktoré upravujú isté vlastností tried, metód, premenných, ...
- static
- final
- abstract
- private
- public
- protected



Zvláštne metódy a premenné?

- `franklin.step(40)`
- `turn(90)` resp. `this.turn(90)`
- volanie metód - kto.čo(upresnenie)

- `Arrays.sort(), Collections.sort()`
- `Arrays.toString(pole)`
- `Math.random(), Math.sin(90)`
- `Integer.parseInt(”145")`
- `Integer.MAX_VALUE, Double.NaN`

static?

- Čo už vieme:
  - funkcionalita je implementovaná v metódach
  - metódy môžeme volať nad objektmi
  - objekty tried vytvárame cez **new**
- Problém z minulosti:
  - kopa metód, ktoré spravia nejakú činnosť, no nie sú nijako **zviazané so stavom objektu** (napr. priamo alebo nepriamo nevyužívajú inštančné premenné)

```
File subor = new File("cisla.txt");
Pomocnik franklin = new Pomocnik();
List<Integer> cisla = franklin.nacitajCisla(subor);
```

static?

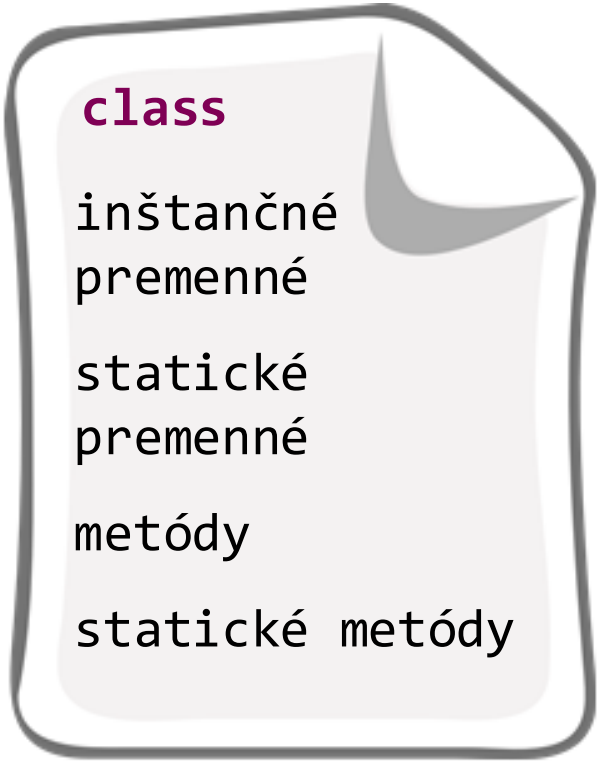
Vytvárame inštanciu len kvôli jednej metóde, ktorá by rovnako dobre fungovala nech by bola v akejkoľvek triede.

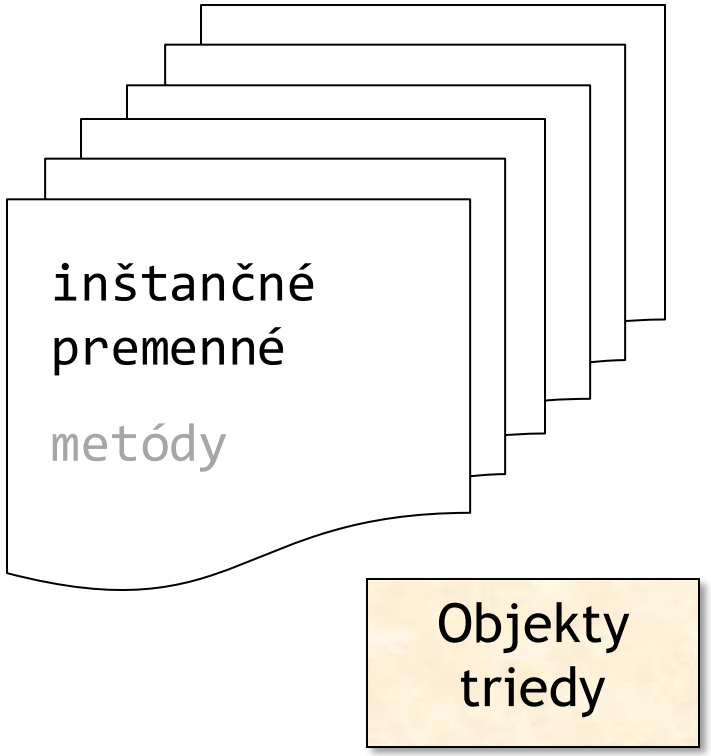
```
File subor = new File("cisla.txt");
Pomocnik franklin = new Pomocnik();
List<Integer> cisla = franklin.nacitajCisla(subor);
```

- Java nemá procedúry a funkcie na rozdiel od:
  - procedurálnych jazykov (Pascal, C, Basic, …)
  - procedurálnych jazykov s OOP rozšírením (Object Pascal, C++, PHP, …)

static

- static = patriaci triede
  - metódy
  - premenné





Statická metóda

```
public class Pomocnik {

  public static List<Integer> nacitajCisla(File subor)
                              throws NacitanieZlyhaloException {

    ...
  }
}
```

V statických metódach nie je **this!**

```
File subor = new File("cisla.txt");
List<Integer> cisla = Pomocnik.nacitajCisla(subor);
```

Statická metóda sa volá **nad triedou**, nie nad referenciou na objekt triedy.

Statická metóda

```
public class Pomocnik {

  public static List<Integer> nacitajCisla(File subor)
                              throws NacitanieZlyhaloException {

    ...
  }
}
```

V statických metódach nie je **this!**

- this (už vieme):
  - `this.` nemusíme (takmer nikdy) písať

Pre pokojnejší život: V statických metódach túto skratku nikdy **nepoužívajte**.

O nepísaní vecí pred .

- `this.` nemusíme písať, lebo Java predpokladá lenivého (ale chytrého) programátora
  - nechytrým sa vypomstí

- Postup, ak Java vidí niečo bez .
  - je to lokálna premenná? – OK, vybavené
  - ak sa doplní `this.`, bude to OK?
    - pozor, v statických metódach `this` neexistuje
  - ak sa doplní názov triedy, bude to OK?
  - inak chyba…

Statické premenné

```
public class Pomocnik {

   private static int pocetVolaniNacitajCisla = 0;

   public static List<Integer> nacitajCisla(File subor)
                              throws NacitanieZlyhaloException {

      Pomocnik.pocetVolaniNacitajCisla++;
      ...
   }
}
```

- K statickým premenným pristupujeme cez názov triedy.
- Statická premenná nemá viacnásobné „inštancie“.

Konštanty

- Zmysluplné využitie statických premenných: konštanty

```
public class PrintedBook {
   public static final String SMALL_FORMAT = "A5";
   private String format;
   ...
}
```

- Kľúčové slovo **`final`** = niečo ako `const` v Pascale
  - Hodnotu `SMALL_FORMAT` už nemožno meniť

```
PrintedBook.SMALL_FORMAT = 7.0;
```

The final field PrintedBook.SMALL_FORMAT cannot be assigned

- Konvencia – VEĽKÉ_PÍSMENÁ

Zrady

- prístup k statickým veciam nestatickým spôsobom:

```
File subor = new File("cisla.txt");
List<Integer> cisla = Pomocnik.nacitajCisla(subor);
```

```
File subor = new File("cisla.txt");
Pomocnik franklin = new Pomocnik();
List<Integer> cisla = franklin.nacitajCisla(subor);
```

Inštančný prístup k statickej veci (len warning, nie chyba) - **nikdy nepoužívame**.

Zrady – pre fajnšmekrov

- prístup k statickým veciam nestatickým spôsobom:

```
File subor = new File("cisla.txt");
List<Integer> cisla = Pomocnik.nacitajCisla(subor);
```

```
File subor = new File("cisla.txt");
Pomocnik franklin = null;
List<Integer> cisla = franklin.nacitajCisla(subor);
```

Je jedno, čo je v premennej `franklin`. Java len použije typ (triedu) premennej.

Statické metódy len rozvažne...

- Statické metódy zvádzajú k lenivosti
  - "Logika": Nechce sa mi vytvárať inštancie, všetko vyhlásim za statické
- Trpíme, lebo globálne zdieľame dáta
- Statické metódy vedú k hroznému návrhu
  - keďže statické metódy nevidia nestatické premenné, vývojár začne zbesilo všetko meniť na statické
- Zmysluplné využitie:
  - pseudotriedy, ktoré sú zoskupením užitočných metód a konštánt

Statické metódy – nič nové

- `java.util.Collections`
  - `Collections.sort(...)`
- `java.util.Arrays`
  - `Arrays.copyOf(...)`
- `java.lang.Math`
  - `Math.min(...)`
- `java.lang.System`
- Mnoho projektov má kopu tried končiacich na `Utils`

main, statické a inštančné spolu

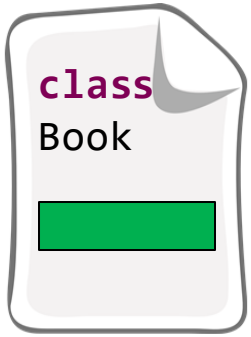
- spúšťacia metóda main je statická
- odporúčané – samostatná metóda v triede Launcher (často sa nazýva aj App)

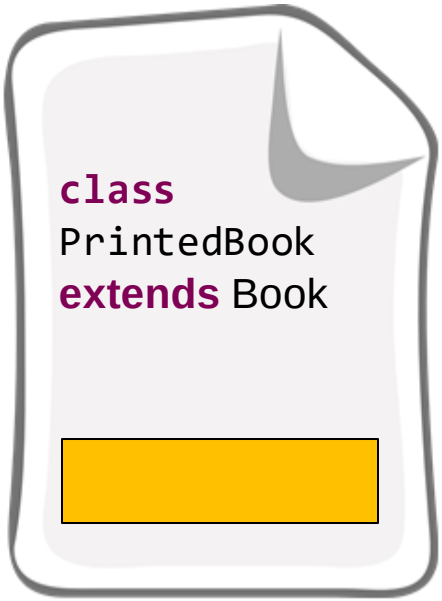
- Musia byť všetky metódy v triede s metódou main tiež statické, aby sa dali použiť?

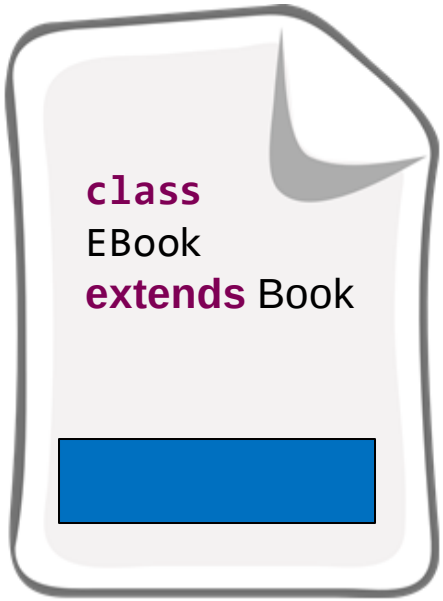
Modifikátor final

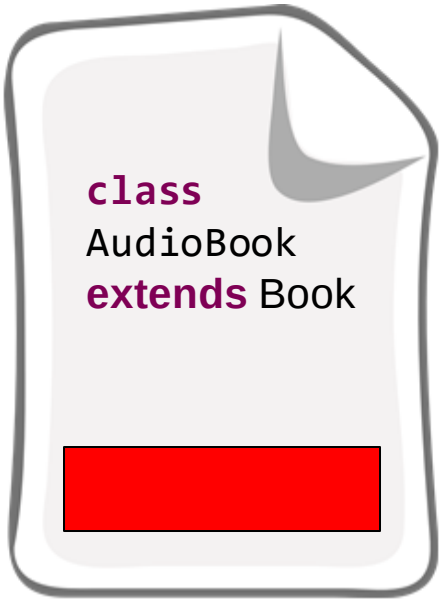
- Keď volám svoju metódu, nemám istotu, že mi ju niekto v rámci rozširovania neprekryl...
  - zvyčajne to chceme dovoliť, ale nie vždy sa to hodí
- Modifikátor **`final`**
  - `final` trieda = zákaz rozširovania
  - `final` metóda = zákaz prekrývania
  - `final` inštančná premenná = hodnotu môžem priradiť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

Spomeňme si


class
Book


class
PrintedBook
extends Book


class
EBook
extends Book


class
AudioBook
extends Book

Problém

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

Abstraktné triedy

```
public abstract class Book {
    ...
}
```

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

Problém

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

- Čo ak tvorca rozširujúcej triedy zabudne metódu `getLocation` 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 Book {
...
  public abstract String getLocation();
...
}
```

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 kníh sú iba objekty takých tried, ktoré majú prekrytú metódu `getLocation()`

```
public class Library {
...
 public void printLocations() {
   for (int i = 0; i < books.length; i++) {
     System.out.print(books[i].getTitle()+": ");
     System.out.println(books[i].getLocation());
   }
 }
...
}
```

Abstraktná trieda vs. interface

- **Interface** – hlavičky metód (**public**)
  - trieda môže implementovať viac rozhraní
  - public hlavičky metód (+ default implementácie), premenné iba public static final
  - použitie – roly, implementujú rôzne nezávislé triedy
- **Abstraktná trieda** – nedá sa vytvoriť inštancia
  - trieda môže rozširovať iba jednu triedu
  - premenné (aj nestatické), metódy (aj nie public) + abstraktné metódy
  - zdieľanie kódu pri súvisiacich triedach (Movie)
  - príklad - AbstractMap

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 vo 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 vo 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...

Zopakujme si

- static
- abstract
- final
- private, (nič), protected, public

- Premenné
- Metódy
- Triedy

Ďakujem za pozornosť !