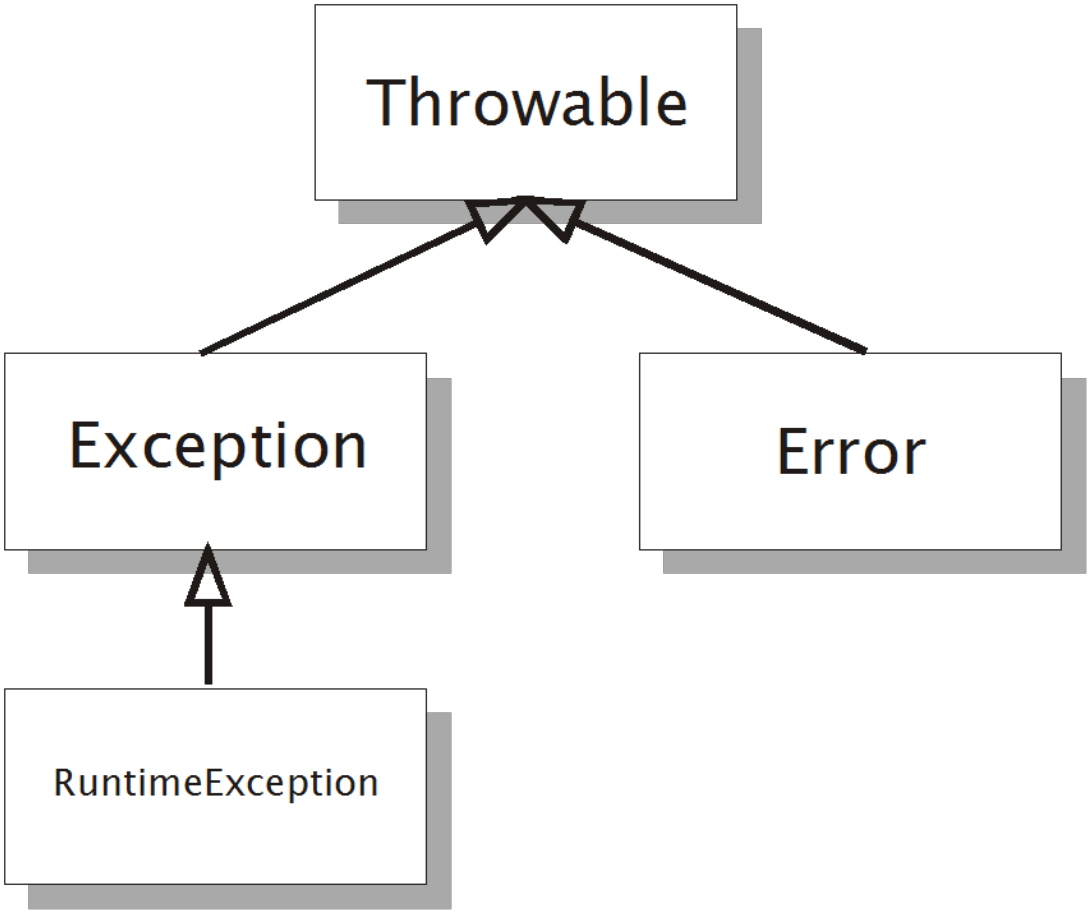
Programovanie, algoritmy, zložitosť (UINF / PAZ1c)

Diel VII.

Róbert Novotný
robert.novotny@upjs.sk
5. 11. 2008

Výnimky a dedičnosť

- výnimky sú triedy v hierarchii



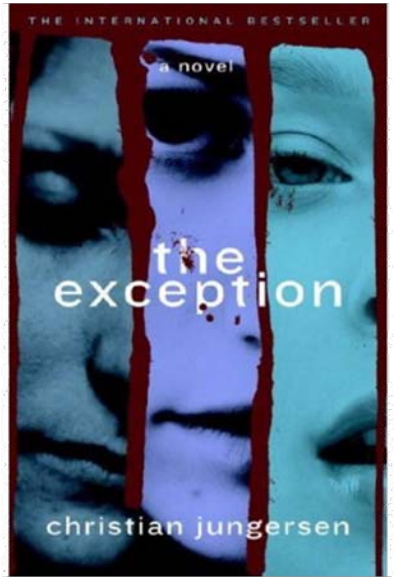
Hierarchia výnimiek v *catch* bloku

```
// Táto metóda vznikla o 4.50 ráno
public void chybnáMetóda {
  try {
    metódaHádžúca IOException();
    metódaHádžúca IllegalArgumentException();
  } catch (IOException e) {
    System.out.println("Nastala výnimka pri čítaní!");
  } catch (Exception e) {
    System.out.println("Nastala všeobecná výnimka! ");
  }
}
```

- výnimka prechádza catch blokmi, kým ju niektorý neodchytí
- prvý *catch* blok odchytí *IOException* a potomkov
- druhý *catch* blok odchytí *Exception* a potomkov

Hierarchia výnimiek v *catch* bloku

- *catch* bloky radíme od najšpecifickejšieho po najkonkrétnejší
  - inak odchytíme výnimku skôr než si želáme
- pozor na hierarchiu: pod výnimku `Exception` spadajú aj nekontrolované výnimky! Neexistuje jednoduchá možnosť, ako odchytiť obyčajné a zvyšné výnimky prebublať



Výnimky pri dedených metódach

```
public class EvidenciaPsov {
  public void zaevidujPsa(Pes pes)
    throws EvidenciaException
  {
    ...
  }
}
```

```
public class SúborováEvidenciaPsov  extends EvidenciaPsov {
  public void zaevidujPsa(Pes pes)
    throws EvidenciaException, IOException
  {
    ...
  }
}
```

Toto nie je povolené. Oddedená metóda môže mať v throws len také isté výnimky ako rodičovská metóda (alebo ich podmnožinu)

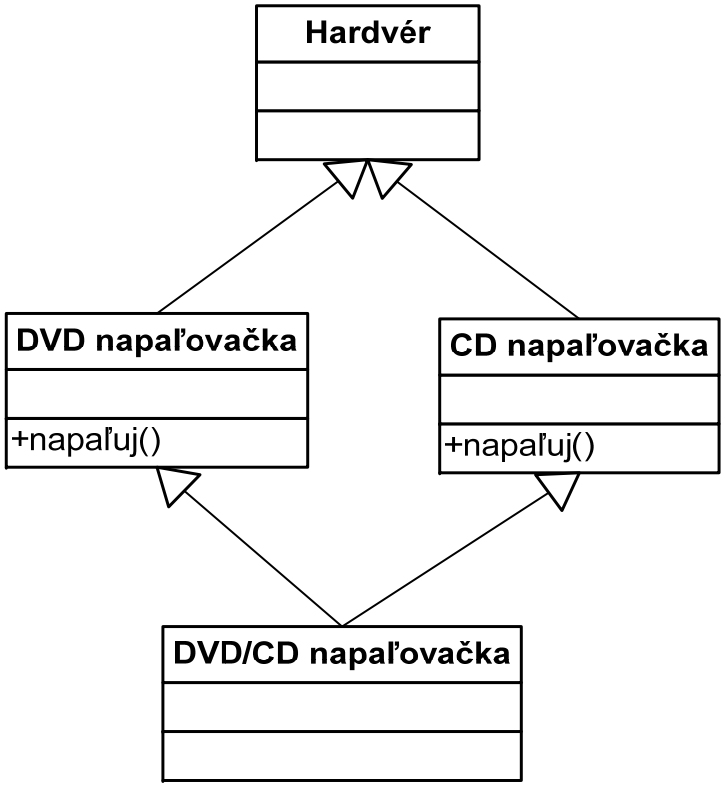
Môžeme mať `throws EvidenciaException` **alebo nič.**

Výnimky pri dedených metódach

- to je ďalší bod kritiky pre kontrolované výnimky
- autor triedy musí veľmi predvídať, aké výnimky budú hádzať potomkovia
- extrémne riešenie: `throws Exception`
  - viď napr. `javax.servlet.HttpServlet`
- samozrejme výnimky v podtriedach možno prebaľovať

Viacnásobná dedičnosť

„Hľadám päť bohatých strýčkov nad hrobom."



- „Smrtiaci smaragd smrti" (*Deadly diamond of Death*)
- DVD/CD napaľovačka zdedí obe metódy na napaľovanie – ale ktorá sa má zavolať kedy?

Viacnásobná dedičnosť

- Nepovedali sme to doslova, ale v Jave neexistuje viacnásobná dedičnosť
- Trieda môže dediť len on jednej triedy
- Čo ak nutne potrebujeme viacnásobnú dedičnosť?

Máme triedu `Pes` a triedu `Spevák`. Chceme spievajúceho psa.

Viacnásobná dedičnosť

- Viacnásobnú dedičnosť vyriešime pomocou rozhraní
  - **rozhranie** (*interface*) a. k. a. *medzitvár* ;-) je trieda, ktorá nemá inštančné premenné a všetky jej metódy sú abstraktné
    - je to teda niečo ako abstraktná trieda
- Trieda môže dediť od viacerých *interfejs*ov

Viacnásobná dedičnosť

- Čo je základnou vlastnosťou speváka?
  - ***spieva*** – teda má metódu `spievaj()`
- Schopnosť spievať môžeme priradiť kde-komu
  - **speváci** (Miro Žbirka!)
  - **herci** (Michal "Zbohom buď, kotlík medený" Dočolomanský)
  - **modelky** (Paris "Každý deň v bulvári" Hilton)
  - **kaderníci** (Martin "Prečítaj list" Kittner)
  - i poniektorí **informatici** (neželajú si byť menovaní)
- Spravíme interface `Spevák` s jedinou metódou `spievaj()`

Viacnásobná dedičnosť

- Čo je základnou vlastnosťou speváka?
  - spieva – teda má metódu `spievaj()`

```
public abstract class Spevák {
   public abstract void spievaj();
}
```

- **V skutočnosti sa však interface definuje**

```
public interface Spevák {
   public void spievaj();
}
```

všetky metódy sú automaticky abstraktné

Viacnásobná dedičnosť

- Ďalej je situácia jednoduchá

```
public SpievajúciPes extends Pes implements Spevák {
   //táto metóda tu MUSÍ byť, prikazuje nám to interface
   public void spievaj() {
       System.out.println("Máám rozpráávkovú búúúdu!");
   }
}
```

- dedenie od rozhraní sa píše pomocou implements
- pravý Javák povie, že „*SpievajúciPes dedí od Pes-a*" a „*implementuje rozhranie Spevák*"

Rozhrania

- trieda môže implementovať ľubovoľný počet rozhraní

```
public ŠaľenyPes extends Pes implements Spevák,
   Comparable, Serializable, Cloneable {
   //..
   //milión metód vyžadovaných od rozhraní
   //...
}
```

Rozhrania

- veselo funguje dedičnosť a polymorfizmus

```
ŠaľenyPes einstein = new ŠaľenyPes();
Pes p = einstein;

Comparable c = new ŠaľenyPes();
Spevák spevák = new ŠaľenyPes();
```

- aj pretypovanie

```
Serializable serializable = new ŠaľenyPes();
Pes pes = (Pes) serializable;
```

- aj `instanceof`

```
ŠaľenýPes einstein = new ŠaľenyPes();
if(einstein instanceof Cloneable) …
```

Rozhrania

- žiadny smrtiaci smaragd smrti - nemáme dilemu, ktorú metódu zavolať
  - *interface* neobsahuje žiaden kód
  - *interface* len hovorí, že táto trieda dokáže túto a túto vec (má túto a túto metódu)
- klasická viacnásobná dedičnosť je napr. v C++
  - interfejsy však riešia problémy smaragdu smrti a málokedy človek skutočne potrebuje dediť od dvoch tried naraz
  - stále vieme zmeniť návrh tried tak, aby sme dedenie od dvoch tried nepotrebovali

Rozhrania

- dilema: kedy použiť abstraktnú triedu a kedy rozhranie?
  - abstraktná trieda je akási šablóna pre potomkov
  - interface definuje skôr rolu, ktorú môže trieda hrať
    - pes vystupuje v role speváka (`implements Spevák`)
    - pes vystupuje v úlohe objektu, ktorý je možné uložiť na disk (`implements Serializable`)
- prax: viac sa používajú rozhrania, než abstraktné triedy.
- vhodná téma na vleklé diskusie a hádky

Rozhrania

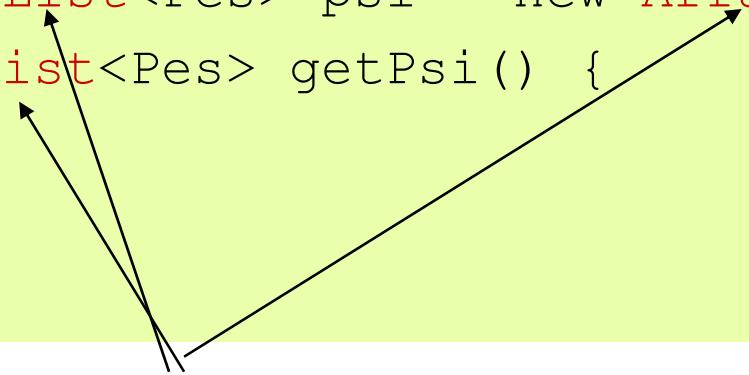
- rozhrania a abstraktné triedy umožňujú hlavne možnosť vymedziť najdôležitejšie vlastnosti
- naviac je možné v prípade potreby zamienať konkrétne *implementácie* rozhrania

```
List<Pes> psi = new ArrayList<Pes>();
```

- ak budeme v parametroch a návratových hodnotách metód používať len interface `List`, môžeme v prípade potreby nahradiť implementáciu pracujúcu nad poľom ľahko nahradiť napr. `LinkedListom`, ktorý pracuje so spojovým zoznamom a nemusíme robiť hromadné nahradenie `ArrayList`-u `LinkedList`-om v tritisíc súboroch nášho projektu.

„Používame interfejsy, nie implementácie!"

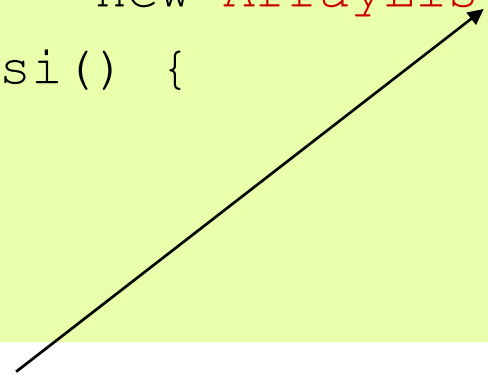
```
public class ChovnáStanica {
  private ArrayList<Pes> psi = new ArrayList<Pes>();
  public ArrayList<Pes> getPsi() {
    return psi;
  }
}
```



- ak chceme zmeniť `ArrayList` (nad poľom) na `LinkedList` (nad spojovým zoznamom) potrebujeme vykonať 3 zmeny len v samotnej triede
- a okrem toho zmeny vo všetkých triedach, ktoré volajú tú metódu (čiže minimálne 1 zmena v testeri)

„Používame interfejsy, nie implementácie!"

```
public class ChovnáStanica {
  private List<Pes> psi = new ArrayList<Pes>();
  public List<Pes> getPsi() {
    return psi;
  }
}
```

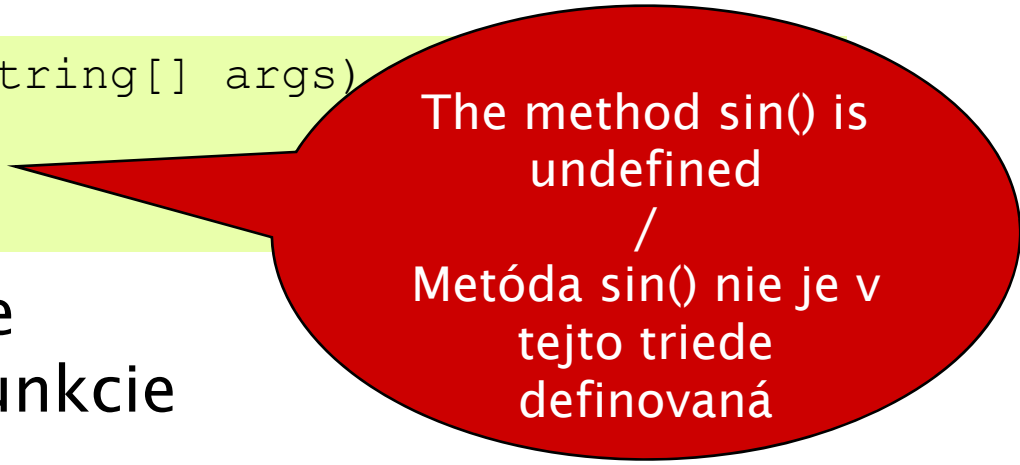


- ak chceme zmeniť `ArrayList` na `LinkedList`, stačí jediná zmena
- ostatné triedy sa nemenia, lebo používajú interfejs

Metódy statické

- Blaise Pascal (1623-1662), autor Turbo Pascalu, chce počítať v Jave sínusy.
- Pokus č. 1:

```
public static void main(String[] args)
  double sinus = sin(25);
}
```



- Spomnime: žiadne voľne poletujúce procedúry/funkcie
- Pokus č. 2: „Aha, trieda `java.lang.Math` má metódu `sin()`!"

```
public static void main(String[] args) {
  Math m = new Math();
  double sinus = m.sin(25);
}
```

Metódy statické

- *„To mám akože kvôli [píííííp] sínusu písať toľko riadkov?"*

```
public static void main(String[] args) {
        double sinus = new Math().sin(25);
}
```

vytvorím inštanciu, zavolám metódu

- Riešenie:
  - Všimnime si hlavičku metódy sin()

```
public static double sin(double a)
```

  - Statické metódy (alias *static methods* alias metódy triedy alias *class methods*) nepotrebujú inštanciu triedy.

```
public static void main(String[] args) {
        double sinus = Math.sin(25);
}
```

metódu volám na triede, nie na inštancii!

Statické importy

- statické importy zjednodušujú zápis

```
import static java.util.Math.*;

public static void main(String[] args) {
  double sinus = sin(25);
}
```

statická metóda je k dispozícii v triede tak, ako keby v nej bola napísaná

- do triedy sa importuje statická metóda `sin()` v triede `Math`
- trieda sa tvári, ako keby v nej bola príslušná statická metóda
- ale: znižuje sa prehľadnosť, užívateľ môže byť zmätený, že má metódu, ktorá nie je v triede

Metódy vs premenné

- Statické metódy sú akoby globálne funkcie/procedúry

- Narušuje sa princíp čistého OOP
  - základom sú objekty. Objektom sa posielajú správy (= metódy)
- Napriek tomu veľmi užitočný koncept
  - základ nejedného návrhového vzoru

Inštančné premenné statické

- Aj inštančné premenné môžu byť statické

```
public class Pes {
   static String farba;
}
```

porušili sme zásadu o getteroch a setteroch

- Načo je to dobré?
- Hodnota statickej premennej je rovnaká pre všetky inštancie. `Typ` nastavujeme všetkým psom sveta.

```
public static void main(String[] args) {
        Pes dunčo = new Pes();
        dunčo.farba = "bledomodrá";

        System.out.println(dunčo.farba);

        Pes lajka = new Pes();
        System.out.println(lajka.farba);
}
```

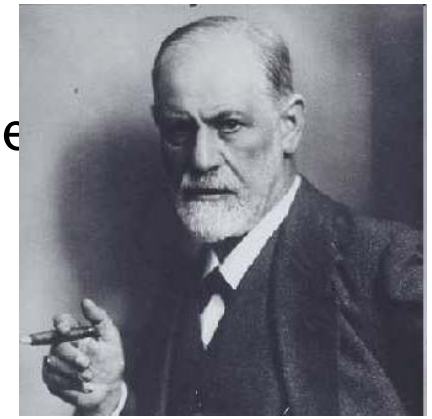
```
bledomodrá
bledomodrá
```

Inštančné premenné statické

```
public class Pes {
        static String farba;
}
```

analógia:
Pes = unit
farba = globálna premenná

- analógia 1 (z procedurálnych jazykov):
  - `Pes` je unit, `farba` je globálna premenná v unite
- analógia 2 (od Freuda)
  - všetky psy majú kolektívnu pamäť
  - typ si zdieľajú medzi sebou



- Ak typ zmeníme na jednom mieste a potom na druhom mieste, zmena sa opäť prejaví na všetkých psoch.

Psi všetkých krajín, nastavte si vek!

Inštančné premenné statické

```
public class Pes {
        static String farba;
}
```

analógia:
Pes = unit
farba = globálna premenná

- Ak farbu zmeníme na jednom mieste a potom na druhom mieste, zmena sa opäť prejaví na všetkých psoch! To môže byť šokujúce a prekvapivé!

```
Pes dunčo = new Pes();
dunčo.farba = "bledomodrá";
System.out.println(dunčo.farba);
Pes lajka = new Pes();
System.out.println(lajka.farba);
lajka.typ = "staroružová";
System.out.println(lajka.farba);
System.out.println(dunčo.farba);
```

```
bledomodrá
bledomodrá
staroružová
staroružová
```

Inštančné premenné statické

```
public class Pes {
        static String farba;
}
```

- Statické premenné prináležia triede. Odporúča sa meniť ich obsahy s použitím triedy:

```
Pes.farba = "khaki";
Pes lajka = new Pes();
System.out.println(lajka.farba);
```

meníme obsah premennej na triede!

meníme farbu všetkým psom sveta!

```
khaki
```

Zmysluplné použitie statických premenných

- Zmysluplné použitie vo svete: konštanty

```
public class MatematickeKonstanty {
	public static final double PI = 3.14;
}
```

- Ak chcem použiť PI, stačí

```
double mojePi = MatematickeKonstanty.PI;
```

- Kľúčové slovo *final* = ,,paródia na const" v Pascale
  - hodnotu v premennej PI už nemožno zmeniť

Metódy vs premenné

- Statické metódy nevidia nestatické inštančné premenné!

```
public class Pes {
  private int vek;
  static void štekaj() {
     for(int i = 0; i < vek; i++) {
       System.out.println("Haf!");
     }
  }
}
```

Cannot make a static reference to a nonstatic field vek
/
Statický prístup k nestatickej premennej!

- Logická otázka + vysvetlenie: keď nemám žiadnu inštanciu, aká je hodnota premennej *vek*?

Metódy vs premenné

- Statické metódy nevidia nestatické inštančné premenné!
- Riešenie: „ta označím `vek` ako `static`!"

```
public class Pes {
  private static int vek;
  static void štekaj() {
     for(int i = 0; i < vek; i++) {
       System.out.println("Haf!");
     }
  }
}
```

- Práve sme zaistili, že zmena veku jedného psa nastaví tento vek aj ostatným psom (žijúcim, aj budúcim)
- Statické metódy sa nesmú spoliehať na stav triedy!

Statické metódy môžu byť zlom!

- statické metódy zvádzajú k lenivosti
  - Logika: *„Nechce sa mi vytvárať inštancie, všetko vyhlásim za statické!“*

```
public class Pes {
  private static int vek;
  static void štekaj() {...}
}
```

```
Pes.vek = 25;
Pes.štekaj();
```

  - a potom budem trpieť, lebo inštancie zdieľajú dáta
- statické metódy vedú ku hroznému návrhu
  - keďže statické premenné nevidia nestatické premenné, vývojar začne zbesilo meniť všetko na statické

Statické metódy môžu byť zlom!

- statické metódy majú problém s viacerými vláknami
  - týka sa paralelného programovania
  - paralelné programovanie má problém, ak sa dáta zdieľajú
  - statické dáta sú zdieľané medzi inštanciami
- zmysluplné využitie:
  - konštanty
  - pseudotriedy, ktoré sú len zoskupením užitočných metód

Zmysluplné využitie

- `java.util.Collections`: trieda plná statických metód užitočných pri kolekciách
  - binárne vyhľadávanie
  - minimum
  - maximum
- analogicky `java.util.Arrays` – práca s poliami
- mnoho projektov má kopu tried končiacich na `Utils`