



12. prednáška (12.12.2016)

**Výnimky (2.časť),
statické metódy a
premenne,
konštanty, jar-y,
JavaDoc**





Čo už vieme o výnimkách

- Pri volaní nejakej metódy môže nastať nejaká výnimka
- Ak výnimka nastane a neodchytíme ju, obdivujeme stack trace

```
Exception in thread "main" java.lang.NullPointerException  
at Vynimkarka.kladnyPriemer(Vynimkarka.java:9)  
at Spustac.main(Spustac.java:10)
```



Čo už vieme o výnimkách

- Ak im vieme predchádzať `if`-mi, predchádzame im `if`-mi

```
public boolean kladnySucet(int[] pole, int k)
{
    if (pole==null) return false;
    if (pole.length==0) return false;
    if (pole.length<k) k=pole.length;
    if (k==0) return false;
    int sucet = 0;
    for (int i = 0; i < k; i++) {
        sucet = sucet + pole[i];
    }
    return (sucet > 0);
}
```



Čo už vieme o výnimkách

- Výnimky sú objekty konkrétnych tried
 - `java.lang.NullPointerException`
 - robíme operáciu typu `null.metóda()`
 - `java.lang.ArithmeticException: / by zero`
 - Delili sme nulou
 - `java.lang.NegativeArraySizeException`
 - `int[] pole = new int[-5];`
 - `java.lang.ArrayIndexOutOfBoundsException: 10`
 - Použili sme index poľa 10, čo je mimo rozsahu poľa, ktoré malo veľkosť 10 alebo menej.



Čo už vieme o výnimkách

- Výnimky, ktoré nevieme ošetriť `if`-mi vieme odchytiť

```
try {  
    // blok príkazov z ktorého odchyťávame výnimky  
} catch (TypVýnimky1 e) {  
    // vysporiadanie sa s daným typom výnimky  
} catch (TypVýnimky2 e) {  
    // vysporiadanie sa s daným typom výnimky  
} finally {  
    // príkazy, ktoré sa vykonajú bez ohľadu na to, čo sa stalo  
}
```



Filozofia výnimiek

- Ak metóda vyhodí výnimku, tak preto, že sa nevie sama vysporiadať s daným stavom
- Je však šanca, že sa s tým vysporiada kus kódu, ktorý túto metódu volal
- Iný pohľad: neviem urobiť, čo si chcel (napr. neviem ti vrátiť hodnotu) a nejako ti to chcem povedať



Filozofia výnimiek

- Dôvod, kedy vyhodit' výnimku:
 - Volajúci kód porušil kontrakt
 - `int f = úbohýObjekt.faktoriál(-20);`
 - `konzola.vykonajPríkaz("Michael Jackson is not dead!");`
 - `človek.setRodnéČíslo("9370888\\0000");`
 - Nie sú podmienky na vykonanie
 - Nemám internet
 - Súbor som nenašiel
 - Zaplnil sa disk
 - Nemám už žiadne dáta, ktoré by som ešte čítal (Scanner)



Výnimky nemusíme nutne odchytať

Táto metóda môže vyhodit'
FileNotFoundException

```
public class Pocitadlo {  
    public List<Integer> nacistajCisla(File f)  
        throws FileNotFoundException {  
        List<Integer> cisla = new ArrayList<Integer>();  
        Scanner citac = new Scanner(f);  
        while(citac.hasNextInt()) {  
            cisla.add(citac.nextInt());  
        }  
        citac.close();  
        return cisla;  
    }  
}
```

Neošetrená výnimka
FileNotFoundException

- Necháme ošetrovanie na niekoho iného



Ako vyhodit' novú výnimku

- Aj my môžeme vyhadzovať výnimky

V tejto metóde môže nastat' výnimka `ZapornyVstupException`

```
public class Pocitadlo {  
    public int faktorial(int n) throws ZapornyVstupException {  
        if (n < 0)  
            throw new ZapornyVstupException();  
        int faktorial = 1;  
        for (int i = 1; i < n; i++) {  
            faktorial = faktorial * i;  
        }  
        return faktorial;  
    }  
}
```

Vyhod' výnimku



Ako vyhodit' novú výnimku

- Aj my môžeme vyhadzovať výnimky
- **Pravidlo:** Každá vyhadzovaná výnimka sa uvádza v bloku **throws**

```
public class Pocitadlo {  
    public int faktorial(int n) throws ZapornyVstupException {  
        if (n < 0)  
            throw new ZapornyVstupException();  
        int faktorial = 1;  
        for (int i = 1; i < n; i++) {  
            faktorial = faktorial * i;  
        }  
        return faktorial;  
    }  
}
```



Častá chyba

- Výnimka má pomôcť pochopiť používateľovi našej metódy, čo sa stalo

```
public int faktorial(int n) throws Exception {  
    ...  
}
```

- Milý programátor. V metóde `faktorial()` môže nastať *nejaký problém*





Ako vyhodit' vlastnú výnimku

- Lepšie riešenie: vytvoríme vhodnejšiu výnimku

```
public int faktorial(int n) throws ZapornyVstupException {  
    ...  
}
```

- Aha! Keď dám záporný vstup, tak mi vyletí výnimka. Dám si na to pozor.



Ako vyhodit' vlastnú výnimku

- Výnimky sú objekty
- Triedy výnimiek môžu mať všetko to, čo ostatné triedy
 - Vlastné inštančné premenné
 - Vlastné metódy
 - Konštruktory
- Vieme teda odovzdať veľmi komplexnú informáciu volajúcemu kódu



Pohadzujeme výnimky

- Ak naša metóda výnimku neošetruje, môže ju posunúť ďalej volajúcej metóde - výnimka vybúbe vyššie
 - Ak výnimku posielame ďalej tak, že ju uvádzame v **throws** bloku hlavičky metódy
 - Platí pravidlo
 - Výnimky musíme **bud' odchytiť** v **catch** bloku
 - Alebo ju môžeme neošetriť a **poslať ďalej**
- “systém padajúceho ...”*
- **Ak nastane problém, niekto by ho vyriešiť mal - nie však používateľ !**



Pohadzujeme výnimky

- Môžeme aj zabaliť výnimku do novej

```
public class Pocitadlo {
    public List<Integer> nacistajCisla(File f)
        throws PocitadloException {
        List<Integer> cisla = new ArrayList<Integer>();
        Scanner citac = null;
        try {
            citac = new Scanner(f);
            while(citac.hasNextInt()) {
                cisla.add(citac.nextInt());
            }
        } catch (FileNotFoundException e) {
            throw new PocitadloException("Čísla nenačítateľné", e);
        } finally {
            if (citac!=null) citac.close();
        }
        return cisla;
    }
}
```



Pohadzujeme výnimky

- Môžeme aj zabaliť výnimku do novej

```
public class Pocitadlo {
    public List<Integer> nacistajCisla(File f) {
        List<Integer> cisla = new ArrayList<>();
        Scanner citac = null;
        try {
            citac = new Scanner(f);
            while(citac.hasNextInt()) {
                cisla.add(citac.nextInt());
            }
        } catch (FileNotFoundException e) {
            throw new PocitadloException("Čísla nenačítateľné", e);
        } finally {
            if (citac != null) citac.close();
        }
        return cisla;
    }
}
```

Hlásenie: Čísla nenačítateľné

Príčina:

FileNotFoundException

Hlásenie: File not found

Vytvoríme novú popisnejšiu výnimku, ktorou obalíme nízkoúrovňovú výnimku



Pohadzujeme výnimky

- Do výnimky `PocitadloException` musíme samozrejme dodať konštruktory (tie sa nededia)

```
public class PocitadloException extends Exception {  
    public PocitadloException() {  
    }  
    public PocitadloException(String message) {  
        super(message);  
    }  
    public PocitadloException(Throwable cause) {  
        super(cause);  
    }  
    public PocitadloException(String message, Throwable cause) {  
        super(message, cause);  
    }  
}
```

príčina výnimky
- iná výnimka



Prebaľovanie výnimiek

- Na čo je dobré prebaľovanie výnimiek ?
- Používateľa nášho programu nezaujíma, čo sa pokazilo v črevách
- *Príklad* : Spadol mi program, bubľajú výnimky:
 - `VSúboreChýbaPremennáException: početBalíčkov`
 - `KonfiguráciaNemáPremennúException: početBalíčkov`
 - `ModulKonfiguráciaOutOfDateException`
 - `StrarýJarSúborException: konfiguracia.jar`
- Mňa nezaujíma prečo nefungoval starý konfiguračný program, stačí, že viem, že potrebujem zohnať novší
- Keby som mal riešiť prvú výnimku, tak neviem čo robiť





Druhy výnimiek

- V Java existujú tri druhy toho, čo vyhadzujeme
 - **Kontrolované** výnimky (Exceptions)
 - Ak metóda hádže výnimku, musí ju uviesť v **throws**
 - Volajúca metóda musí výnimku odchytiť, alebo ju tiež uviesť v **throws**
 - **Nekontrolované** výnimky (Runtime Exceptions)
 - Ak metóda hádže výnimku, nemusí ju uviesť v **throws**
 - Potomkovia triedy `RuntimeException`
 - **Chyby** (Errors)
 - Rovnaký princíp ako nekontrolované výnimky
 - Potomkovia triedy `Error`



Chyby

- Chyby: abnormálny stav systému, aplikácia nemá šancu sa zotaviť
 - `OutOfMemoryError`: došla pamäť
 - `VirtualMachineError`: virtuálny stroj nevie púšťať programy
- Vytvárať ich má zmysel, aby sa dalo zistiť miesto a príčina chyby





Druhy výnimiek

- Ktorý typ výnimky použiť?

„Kontrolované výnimky sú experimentom, ktorý zlyhal.“

- Bruce Eckel, hrdina Javy



„Kontrolované výnimky pre zotaviteľné chyby, nekontrolované pre programátorské chyby.“

- Joshua Bloch, iný hrdina Javy



- žiadny iný OOP jazyk nemá kontrolované výnimky
 - ani C# (poučili sa(?)), ani Python, ani C++...



Filozofia Eckela a spol.

- Všetky výnimky vyrobíme ako nekontrolované **ALE**:
 - Uvedieme ich do dokumentácie
 - Uvedieme ich do **throws** (ak to má zmysel)
- **Dokumentácia**: nenastane výnimka, ktorú nik nečakal
 - nič nečakané nevybude z vnútra čriev cudzieho kódu
- **Throws**:
 - Eclipse môže automaticky generovať **catch** bloky
 - Aj keď nie je dokumentácia, máme informáciu o možných výnimkách



Výnimky pri prekrývaní metód

```
public class Film {
public void dajUmiestnenie() throws FilmException {
    ...
}
}
```

```
public class FilmNaDvd extends Film {
public void dajUmiestnenie() throws
    FilmException, DvdException {
    ...
}
}
```

Môžeme mať throws
FilmException alebo NIČ

Toto nie je pre kontrolované
výnimky povolené!
Prekrývajúca metóda môže
mať v throws iba podmnožinu
výnimiek pôvodnej triedy



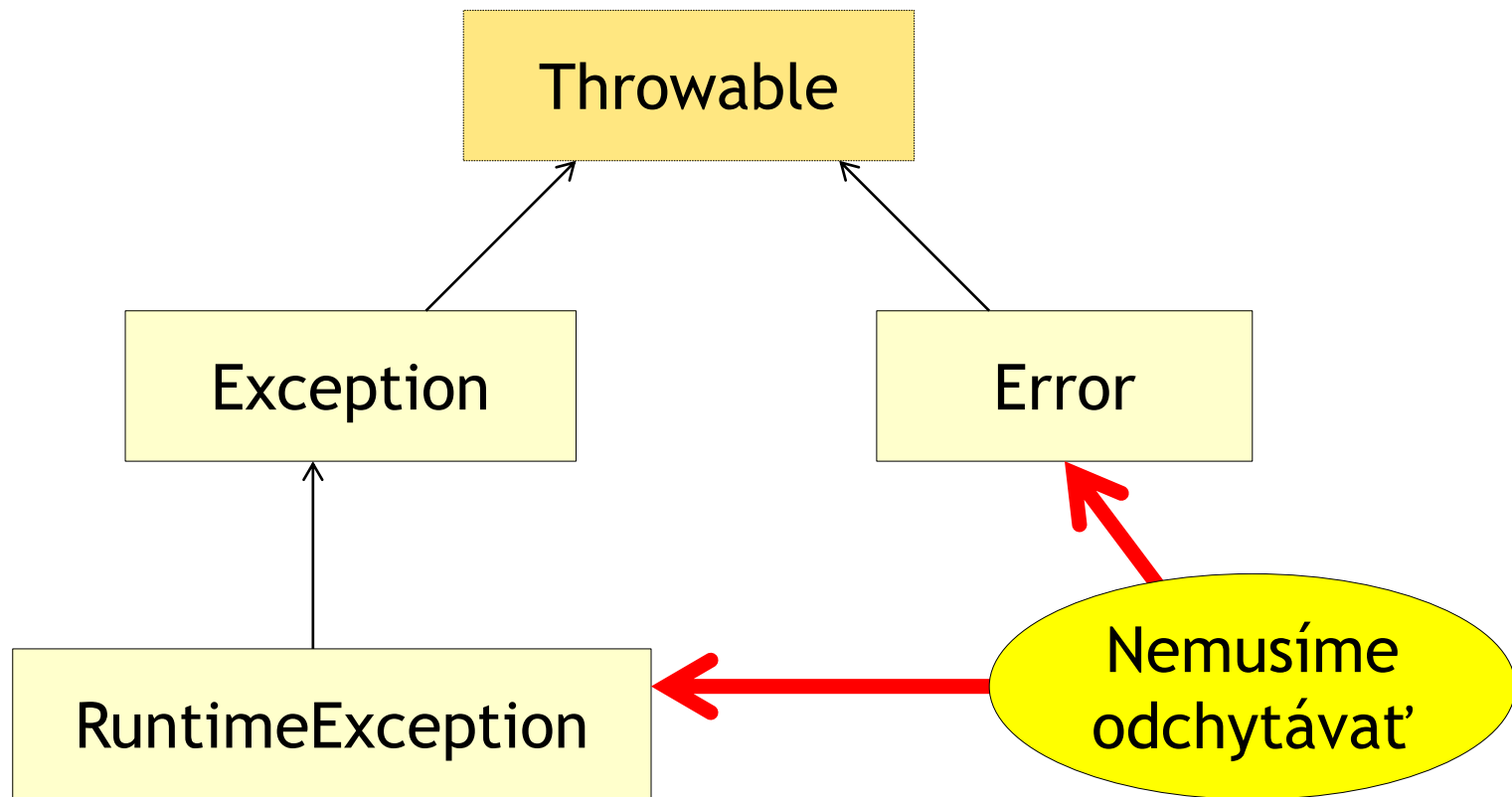
Výnimky pri prekrývaní metód

- Súčasť kritiky kontrolovaných výnimiek
- Autor rodičovskej triedy musí predvídať, aké výnimky budú hádzat' potomkovia



Výnimky v hierarchii

- Výnimky sú triedy v hierarchii dedičnosti





Hierarchia výnimiek v *catch* blokoch

- Výnimky sú triedy v hierarchii dedičnosti

```
try {  
    ...  
} catch (FileNotFoundException e) {  
    System.err.println("Nenašiel som súbor");  
} catch (IOException e) {  
    System.err.println("Vstupno-výstupná chyba");  
} catch (Exception e) {  
    System.err.println("Nastala nejaká výnimka");  
}
```

- Výnimka prechádza **catch** blokmi, pokiaľ ju niektorý neodchytí
- Prvý **catch** blok odchytí `FileNotFoundException` a potomkov
- Druhý **catch** blok odchytí `IOException` a potomkov
- Tretí **catch** blok odchytí `Exception` a potomkov



Hierarchia výnimiek v *catch* blokoch

- **catch** bloky radíme od najšpecifickejšieho po najvšeobecnejší
 - Inak odchytíme výnimku skôr, ako si želáme
- Pozor na hierarchiu: pod výnimkou `Exception` sú aj nekontrolované výnimky `RuntimeException`
 - Neexistuje jednoduchá možnosť ako odchytit' iba kontrolované výnimky a nekontrolované poslať vyššie



Výnimky - časté chyby

- Problém neriešime - Všetko zatajíme - výnimka sa zhltné
 - Keď program zdochne, nik nevie prečo a kde

```
try {  
    citac = new Scanner(f);  
} catch (FileNotFoundException e) {}
```

- Banality riešime výnimkami

```
try {  
    int i = 0;  
    while (true) {  
        pole[i+1] = 2 * pole[i];  
        i++;  
    }  
} catch (ArrayIndexOutOfBoundsException e) {}
```



Výnimky - časté chyby

- Nechce sa nám robiť zmysluplné výnimky

```
void metóda() throws Exception {  
    ...  
}
```

- Neprebalené výnimky bublajú príliš vysoko
 - Sťažujeme sa na veci, ktoré už volajúci kód určite nevyrieši

```
void upečKoláč() throws IOException, SQLException {  
    ...  
}
```



Výnimky - záver

- Výnimky slúžia na prípady, keď sa metóda nevie s danou situáciou vysporiadať sama
- Ostatné výnimočné situácie riešime priamo v metóde (**if**, **while**, **try-catch**)



Statické metódy

- Príde ničnetušiaci Pascalovský programátor vyskúšať Javu, a chce vypočítať sínus:

```
public static void main(String[] args) {  
    double sinus = sin(3.14);  
}
```

Method sin() is
undefined

- *“To je čo za jazyk, čo nemá sínus?”*
- V Jave nie sú žiadne voľne poletujúce metódy a globálne premenné



Statické metódy

- “Aha, trieda `java.lang.Math` má metódu `sin()`”

```
public static void main(String[] args) {  
    Math m = new Math();  
    double sinus = m.sin(3.14);  
}
```

- “To mám akože kvôli [píííp] sínusu vyrábať nejaký objekt na jedno použitie?”
- Lenže my mu už vieme poradiť, že to robiť nemusí:

```
public static void main(String[] args) {  
    double sinus = Math.sin(3.14);  
}
```




Statické metódy

- Všimnime si hlavičku metódy `sin`

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

- Statické metódy nepotrebuju inštanciu triedy
 - Známe aj ako metódy triedy
- *“Statické metódy sú akoby globálne funkcie/procedúry”*
- Narušuje sa princíp OOP
 - Základom OOP sú objekty, na ktorých voláme metódy



Statické "inštančné" premenné

- Aj inštančné premenné môžeme zmeniť na statické

```
public class FilmNaDvd {
    static double polomerDvd;
}
```

Porušili sme zásadu
o priamom prístupe

- Načo je to dobré?
- Hodnota statickej premennej je spoločná pre všetky inštanacie

```
public static void main(String[] args) {
    FilmNaDvd matrix = new FilmNaDvd();
    matrix.polomerDvd = 5.5;
    FilmNaDvd shawshank = new FilmNaDvd();
    System.out.println(matrix.polomerDvd);
    System.out.println(shawshank.polomerDvd);
}
```

5.5
5.5



Statické "inštančné" premenné

- Ak zmeníme statickú premennú cez referenciu na jeden objekt, prejaví sa to vo všetkých objektoch
 - Kto zabudol, že ide o statickú premennú je zmätený

```
public static void main(String[] args) {  
    FilmNaDvd matrix = new FilmNaDvd();  
    matrix.polomerDvd = 5.5;  
    FilmNaDvd shawshank = new FilmNaDvd();  
    System.out.println(matrix.polomerDvd);  
    System.out.println(shawshank.polomerDvd);  
    shawshank.polomerDvd = 3.9;  
    System.out.println(matrix.polomerDvd);  
    System.out.println(shawshank.polomerDvd);  
}
```

```
5.5  
5.5  
3.9  
3.9
```



Statické “inštančné” premenné

- Statická premenná je **spravovaná v triede**, nie v objektoch!
- Hodnota nie je uložená v inštanciách, ale v triede
- Nesprávne (ale žiaľ uskutočniteľné) použitie:
 - `matrix.polomerDvd = 6.0;`
- Správne pristupujeme cez triedu:
 - **`FilmNaDvd`**.`polomerDvd = 6.0;`

Meníme polomer
všetkým DVD-čkám



Konštanty

- Zmysluplné využitie statických premenných: konštanty

```
public class FilmNaDvd {  
    public static final double POLOMER_DVD = 6.0;  
    private String nazovFilmu;  
    ...  
}
```

- Klúčové slovo **final** = niečo ako `const` v Pascale
 - Hodnotu `POLOMER_DVD` už nemožno meniť

```
FilmNaDvd.POLOMER_DVD = 6.0;
```

The final field
`FilmNaDvd.POLOMER_DVD`
cannot be assigned



Statické metódy vs. premenné

- Statické metódy nevidia nestatické inštančné premenné

```
public class FilmNaDvd {  
    private double String nazovFilmu;  
    static void vypisNazov()  
        System.out.println(nazovFilmu);  
}  
}
```

Cannot make a static reference to a nonstatic field nazovFilmu

- Ked' nemám inštanciu, aká je hodnota názvu filmu?



Statické metódy vs. premenné

- Statické metódy nevidia nestatické inštančné premenné
 - “riešenie”: Označíme `nazovFilmu` ako `static`

```
public class FilmNaDvd {  
    private static double nazovFilmu;  
    static void vypisNazov()  
        System.out.println(nazovFilmu);  
    }  
}
```

- `nazovFilmu` bude odteraz rovnaký pre všetky inštancie
- Statické metódy sa nesmú spoliehať na premenné inštancií



Statické metódy môžu byť veľké zlo

- Statické metódy zvädzajú k lenivosti
 - “Logika”: Nechce sa mi vytvárať inštancie, všetko vyhlásim za statické
- Trpíme, lebo inštancie zdieľajú 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



Známe pseudotriedy so statickými metódami

- `java.util.Collections`
- `java.util.Arrays`
- `java.lang.Math`
- `java.lang.System`
- **Mnoho projektov má kopol tried končiacich na
Utils**



- Dokumentácia je súčasťou každého slušného projektu
- Do dokumentácie sa zahrnú komentáre pred triedami, inštančnými premennými a metódami, ktoré začínajú znakmi / * *
- Komentáre metód majú aj niekoľko špeciálnych označení
 - `@param vstupny_parameter` popis vstupného parametra
 - `@return` popis výstupnej hodnoty
 - `@throws` vymenovanie vyhadzovaných výnimiek



- Generujeme dokumentáciu

1. Pravým na projekt > export.. > Java > **JavaDoc**
2. Nastavíme od akých modifikátorov viditeľnosti sa má dokumentovať
 - Ak dáme napríklad protected tak dokumentované bude všetko protected a public
3. Finish

- Ďalšie čítanie:

- <http://java.sun.com/j2se/javadoc/writingdoccomments/>



Spúšťame Javu z príkazového riadka

- Keď ešte neboli balíčky, nabehli sme do adresára, kde bol Spustac.class a stačilo spustiť

```
cmd> cd projektXX/bin  
cmd> java Spustac
```

- Čo však keď, sme šli napr. o adresár vyššie?

```
cmd> cd ..  
cmd> java bin/Spustac
```

Exception in thread "main"
java.lang.NoClassDefFoundError:
bin/Spustac (wrong name: Spustac)



Spúšťame Javu z príkazového riadka

- Java hľadá v adresároch uvedených v premennej prostredia CLASSPATH
 - **WinXP:** Ovládacie panely > Systém > Upresniť > Premenné prostredia > Systémové premenné
 - **Win7:** Computer > System properties > Advanced System Settings > Advanced > Enviroment Variables
 - **Unix a podobné:**
 - CLASSPATH=cesta1:cesta2:...
 - export CLASSPATH
- Adresáre alebo jar alebo zip súbory v CLASSPATH tvoria spolu korene balíčkovej hierarchie



Spúšťame Javu z príkazového riadka

- Ak sme dodali náš bin adresár do CLASSPATH, môžeme **hocikde** na disku iba napísať:

```
cmd> java Spustac
```

- Pred tým to šlo, pretože ak neexistuje CLASSPATH, tak **do CLASSPATH sa automaticky pridáva aj aktuálny adresár**
- CLASSPATH môžeme rozšíriť aj hneď pri spúšťaní

```
cmd> java -cp cesta_ku_bin Spustac
```



Spúšťame Javu z príkazového riadka

- Môžeme dodať aj jar súbory

```
cmd> java -cp cesta_ku_bin:jpaz2.jar Spustac
```

- Ak je náš spúšťač v balíčku musíme napísať úplný názov triedy aj s balíčkom

```
cmd> java -cp cesta_ku_bin  
sk.upjs.paz11.Spustac
```

V CLASSPATH musí byť cesta ku koreňu balíčkovvej hierarchie



Spúšťame Javu z príkazového riadka

- Ak vlezeme do balíčka, ktorého koreň nie je v CLASSPATH nevieme spúšťať lokálne class súbory!

```
cmd> cd cesta_ku_bin/sk/upjs/paz11  
cmd> java Spustac
```

Exception in thread "main"
java.lang.NoClassDefFoundError:
Spustac (wrong name: Spustac)

- **VŽDY** musíme uviesť úplné meno triedy



Generujeme jar súbory

- Spúšťanie cez príkaz java nie je nijako pohodlné, najmä, ak potrebujeme použiť mnoho externých knižníc
- Čo je horšie, máme mnoho relatívne malých tried
- Pri integrovaní viacerých projektov môže vzniknúť neprehľadná spleť class súborov
- Riešením je vytváranie jar súborov



Čo je jar?

- Jar je skratka od Java archive
- Reálne je to obyčajný zip súbor so zbalenou hierarchiou class súborov a prípadných ďalších vecí
- Navyiac obsahuje súbor META-INF/MANIFEST.MF v ktorom sú prídavné informácie
 - Využijeme na nastavenie hlavnej spustiteľnej triedy



Generujeme jar súbory

- Samotné generovanie necháme na Eclipse
- Ak chceme, môžeme dobaľiť aj iné jar súbory na ktorých sme závislí (jpaz2.jar), potrebujeme na to ale interné knižnice:
 1. vytvor si v projekte adresár lib a nakopíruj si doňho potrebné jar-y (lib je v adresári projektu vedľa src)
 2. refreshni projekt (F5) - v libe sa objavia jar-y
 3. pravým klik na projekt>build path>configure build path
 4. v záložke library si odstránime externé závislosti
 5. Cez "add jar" pridáme interné knižnice



Generujeme jar súbory

- Generujeme jar

1. pravým na projekt > export.. > Java > **Runnable JAR files** ak chceme spustiteľné alebo **JAR files** ak spustiteľné nechceme
2. Ak Runnable: v Launch configuration treba vybrať triedu ktorá sa má spúšťať
3. V export destination si zvolíme názov a umiestnenie jar súboru
4. Ak chceme dobaľiť interné jar knižnice: Vybrať možnosť "Package required libraries into generated JAR"
5. Môžeme si uložiť ANT skript, aby sme nabudúce nemuseli klikat' tento wizard a mohli rovno pustiť skript
6. Finish



Spúšťame jar z príkazového riadka

- Spustenie spustiteľného jar súboru:

```
cmd> cd adresar_kde_je_jar  
cmd> java -jar subor.jar
```

- Môžeme zároveň rozširovať CLASSPATH

```
cmd> cd adresar_kde_je_jar  
cmd> java -cp jpaz2.jar -jar subor.jar
```



Ďakujem za pozornosť !

