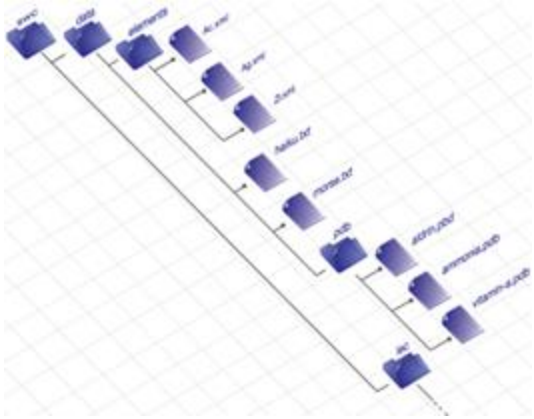
13. prednáška

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

Vlastné výnimky a zopár užitočných vecí



alebo

Koniec PAZ1a

Výnimka

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

Čo sú to výnimky?

- **Výnimky**
  - **špeciálne objekty** výnimkových tried
  - vznikajú vo **výnimočných stavoch**, keď nejaké metódy nemôžu prebehnúť štandardným spôsobom alebo nevedia vrátiť očakávanú hodnotu
  - takmer všetky moderné programovacie jazyky signalizujú výnimočný (neočakávaný) stav vo forme výnimiek

ArrayIndexOutOfBoundsException

Class ArrayIndexOutOfBoundsException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IndexOutOfBoundsException
          java.lang.ArrayIndexOutOfBoundsException

All Implemented Interfaces:

Serializable

```
public class ArrayIndexOutOfBoundsException
extends IndexOutOfBoundsException
```

Thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array.

Rôzne triedy výnimiek

- `java.lang.NullPointerException`
  - robíme operáciu typu `null.metoda()`
- `java.lang.ArithmeticException: / by zero`
  - delili sme celočíselne 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
- `java.io.FileNotFoundException`
  - pokúšame sa otvoriť súbor na čítanie, ktorý neexistuje, alebo zapisovať do súboru na mieste, kde sa to nedá

Výnimková skaza

- Hodená výnimka **okamžite ukončuje** každú metódu alebo blok príkazov, kde sa vyskytne, a postupne vybubláva
  - ak tomu nezabránime...

try-catch-finally

```
try {
  // ...
} catch (TypVýnimky1 e) {
  // ...
} catch (TypVýnimky2 e) {
  // ...
} finally {
  // príkazy, ktoré sa vykonajú bez ohľadu na to,
  čo sa stalo
}
```

Rozdelenie výnimiek:

- **kontrolované** – checked: musia sa odchytávať
- **nekontrolované** – unchecked: nemusia sa odchytávať

Klasika...

```
public class Pomocnik {

  public List<Integer> nacitajCisla(File subor) {
    try (Scanner sc = new Scanner(subor)) {
      List<Integer> vysledok = new ArrayList<>();
      while (sc.hasNext()) {
        vysledok.add(sc.nextInt());
      }
      return vysledok;
    } catch (FileNotFoundException e) {
      System.err.println("Chyba");
    }

    return null;
  }
}
```

Musíme chytať, lebo kontrolovaná výnimka, iné výnimky vybublajú...

Kde je `finally` so zatvorením `Scannera`?

try-catch-finally

```
try {
  // ...
} catch (TypVýnimky1 e) {
  // ...
} catch (TypVýnimky2 e) {
  // ...
} finally {
  // príkazy, ktoré sa vykonajú bez ohľadu na to,
  čo sa stalo
}
```

Rozdelenie výnimiek:

- **kontrolované** - checked: ~~musia sa odchytávať~~
- **nekontrolované** - unchecked: nemusia sa odchytávať

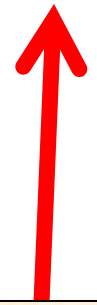
Kontrolované výnimky

- Filozofický pohľad:
  - zotaviteľné chyby
- Programátorsky pohľad:
  - výnimky, ktoré sa nemôžu „šíriť“ bez povšimnutia

- Kontrolované výnimky **nemusíme** odchytávať, ale **nesmú byť vyhodené** z metódy bez toho, aby to mala metóda vo svojom popise...

Klasika...

```
public class Pomocnik {

  public List<Integer> nacitajCisla(File subor) throws
                                          FileNotFoundException {
    try (Scanner sc = new Scanner(subor)) {
      List<Integer> vysledok = new ArrayList<>();
      while (sc.hasNext()) {
        vysledok.add(sc.nextInt());
      }
      return vysledok;
    }
  }
}
```



Môžeme nechať „vybublať“ aj kontrolovanú výnimku, len to musíme explicitne uviesť

throws

```
public … metoda(…) throws TypVýnimky1, TypVýnimky2, ... {

}
```

Zoznam typov výnimiek (výnimkových tried), ktoré môže metóda vyhodiť ako upozornenie pre používateľov metódy.

- v throws:
  - **môžu byť** uvedené **nekontrolované** výnimky, ktoré sú z metódy vyhadzované („vybublávané“)
  - **musia byť** uvedené **kontrolované** výnimky, ktoré sú z metódy vyhadzované („vybublávané“)

Kontrolovaná vs. nekontrolovaná

Ako zistiť, ktorá výnimka je kontrolovaná a ktorá je nekontrolovaná?

Rodokmeň výnimiek

Class ArrayIndexOutOfBoundsException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IndexOutOfBoundsException
          java.lang.ArrayIndexOutOfBoundsException

Class OutOfMemoryError

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.OutOfMemoryError

Class FileNotFoundException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException

Throwable

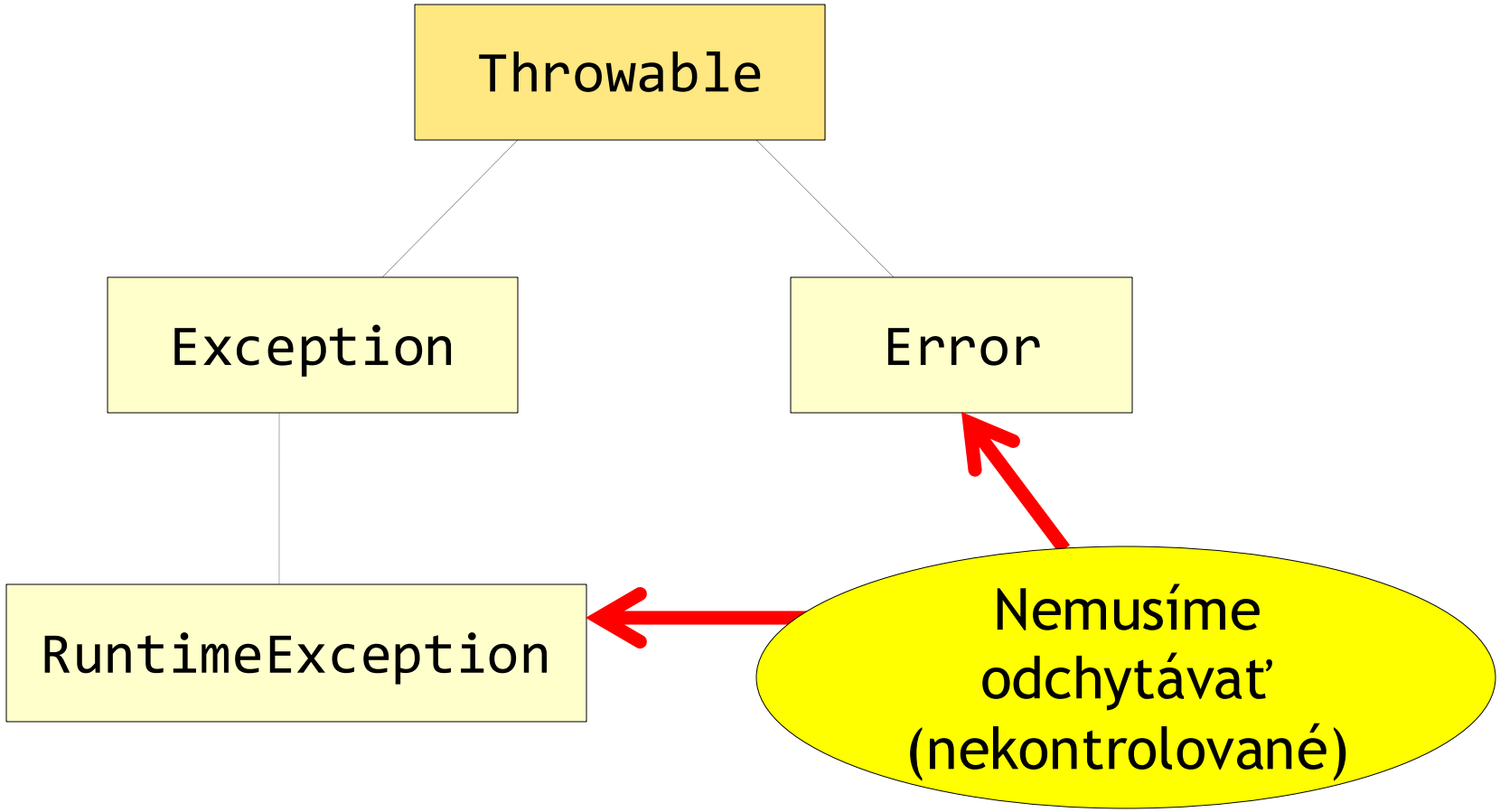
- Každý objekt, ktorý **sa dá vyhodiť**, musí byť inštanciou triedy, ktorá rozširuje triedu `Throwable`
  - upozornenie: koncovkou „able“ končia zvyčajne mená rozhraní (`Iterable`, `Comparable`, `Runable`, …), `Throwable` je ale trieda…

Class FileNotFoundException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException

Výnimky v hierarchii

- Výnimky sú triedy v hierarchii dedičnosti



Druhy výnimiek

- Nekontrolované výnimky (Runtime Exceptions)
  - nemusia sa uviesť v **`throws`**
  - potomkovia triedy `RuntimeException`
- Chyby (Errors)
  - ako nekontrolované výnimky...
  - abnormálny stav systému, aplikácia nemá šancu sa zotaviť
  - vznikajú, aby sa dalo zistiť miesto a príčina chyby
  - potomkovia triedy `Error`
- Kontrolované výnimky (Exceptions)
  - musia sa uviesť v **`throws`**
  - čokoľvek, čo nie je nekontrolovaná výnimka alebo chyba
  - zvyčajne rozširuje triedu `Exception`

Ktorá je aká?

Class ArrayIndexOutOfBoundsException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.lang.RuntimeException
        java.lang.IndexOutOfBoundsException
          java.lang.ArrayIndexOutOfBoundsException

Class OutOfMemoryError

java.lang.Object
  java.lang.Throwable
    java.lang.Error
      java.lang.VirtualMachineError
        java.lang.OutOfMemoryError

Class FileNotFoundException

java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.io.FileNotFoundException

Vyhadzovanie výnimiek

```
if (subor == null) {
  throw new IllegalArgumentException(
                        "Parameter subor nesmie byt null.");
}
```

Cez príkaz **throw** vyhodíme referenciu na objekt výnimkovej triedy.

Constructors

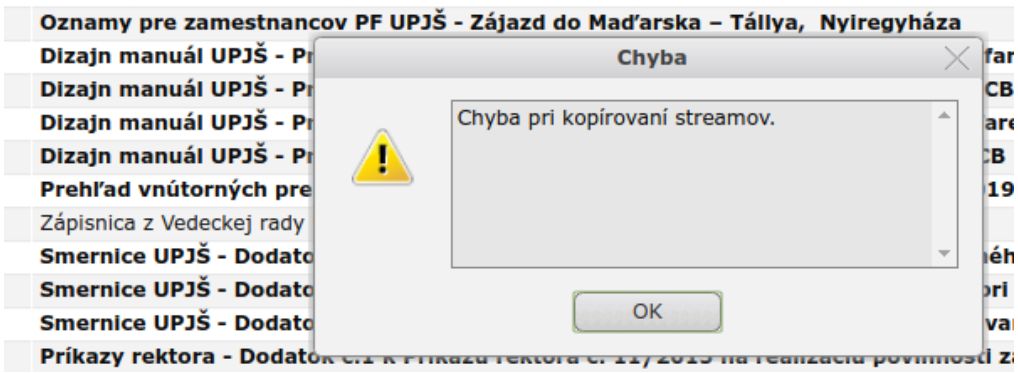
Constructor and Description
IllegalArgumentException() Constructs an IllegalArgumentException with no detail message.
IllegalArgumentException(String s) Constructs an IllegalArgumentException with the specified detail message.

Informačná hodnota výnimky

- Cieľ: načítať čísla zo súboru
- Odchytená výnimka: `InputMismatchException`

Používateľa nezaujíma, čo sa deje vo vnútri metódy a aká interná výnimka vznikla.

```
Chcem kávu:
PrazdnyZasobnikC10Exception
```



Informačná hodnota výnimky

- Výnimka má poskytnúť informácie tak, aby používateľ metódy (=iný programátor) **vedel čo najskôr identifikovať príčinu** alebo zostaviť dobrý „odchytávaco-reakčný“ kód
  - výnimky vyhodené počas vývoja
  - výnimky vyhodené pri testovaní
  - výnimky zaznamenané v logoch pri behu aplikácie
- Zdroje informácii:
  - názov výnimkovej triedy
  - popisná správa vo výnimke
  - výnimka nižšej úrovne (príčina)

Vlastné výnimkové triedy

- Vytvoríme triedu rozširujúcu `Exception`, `RuntimeException` alebo inú existujúcu výnimkovú triedu
  - vlastné zmysluplné konštruktory
  - málokedy: môžeme pridať vlastné inštančné premenné alebo metódy
- Vlastné konštruktory
  - konštruktory sa nededia
  - zvyčajne sa inšpirujeme konštruktormi rodičovskej triedy

Vlastné výnimkové triedy

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

príčina výnimky
- iná výnimka

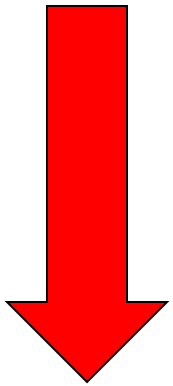
Vyhadzujeme vlastnú výnimku

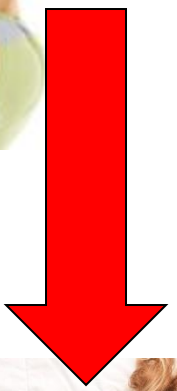
```
public List<Integer> nacitajCisla(File subor)
                        throws NacitanieZlyhaloException {

  …

  if (!subor.exists()) {
    throw new NacitanieZlyhaloException(
                        "Subor " + subor + " neexistuje.");

  …

}
```

Prebaľovanie výnimiek

- Odchytenú výnimku nižšej úrovne „prebalíme“ do vlastnej výnimky s väčšou informačnou hodnotou.

```
FileNotFoundException
InputMismatchException
…
```





```
NacitanieZlyhaloException
```

Prebaľovanie výnimiek

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

  …
  try (Scanner sc = new Scanner(subor)) {
  …
  …
  } catch (InputMismatchException e) {
    throw new NacitanieZlyhaloException(
                              "Subor obsahuje neciselny retazec.", e);
  }
}
```

Vyhadzujeme prebalenú výnimku

Príčina (cause) vyhodenej výnimky

`catch` bloky – ako to naozaj je

- 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

`catch` bloky – ako to naozaj je

- **`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 odchytiť iba kontrolované výnimky a nekontrolované poslať vyššie

Výnimky pri prekrývaní metód

```
public class Book {
public void getLocation() throws BookException {
        ...
}
   }
```

```
public class AudioBook extends Book {
public void getLocation() throws
                BookException, AudioBookException {
        ...
}
```

Môžeme mať `throws BookException` alebo NIČ

Toto je pre **kontrolované výnimky** zakázané!
Prekrývajúca metóda môže mať v throws iba podmnožinu kontrolovaných výnimiek pôvodnej triedy

Kontrolované vs. nekontrolované

- Ktorý typ výnimky použiť?

  „Kontrolované výnimky sú experimentom, ktorý zlyhal.“

  – Bruce Eckel

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

  – Joshua Bloch

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

Výnimky - časté chyby

- Problém neriešime - všetko zatajíme – výnimka sa zhltne
  - program nebeží, ale nik nevie prečo...

```
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 {
    ...
}
```

ZOPÁR BONUSOV

switch

- namiesto `if-elseif-elseif-elseif-...-elseif-else`
- aplikovateľné na: byte/Byte, short/Short, char/Character, int/Integer, String, enum
- menu v konzole s využitím Scanner-a a System.in

```
switch (value) {
        case 1:
                // nejaký kód
                break;
        default:
                break;
}
```

if (value == 1)

else

Anonymné triedy

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

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

- Objekty na jedno použitie (pozor, nie je to static)
- Anonymná trieda
  - nemá názov
  - je definovaná priamo pri vytváraní inštancie

demo

O čom je dobré vedieť

- Trieda v triede (nested, inner, local classes)
- Number literals (ako zapísať čísla v dvojkovej sústave a pod.)
- Format numbers (printf)
- Enum (zopár možností)
- Java Style Guide (čo sa patrí pri písaní kódu)
- Ternary operator (?, :)
- ...

Effective Java

- *Best practices for the Java Platform*
- Avoid creating unnecessary objects
- Prefer try-with-resources to try-finally
- Always override hashCode..., toString
- Minimize mutability
- Prefer interfaces to abstract classes
- Design method signatures carefully
- Return empty collections or arrays, not nulls
- Write doc comments for all exposed API elements

Effective Java

- Minimize the scope of local variables
- Prefer for-each loops to traditional for loops
- Avoid float/double if exact answers are required
- Prefer primitive types to boxed primitives
- Avoid strings where other types are more appropriate
- Beware the performance of string concatenation
- Refer to objects by their interfaces
- Don't ignore exceptions

Rýchlejší ako scanner

- BufferedReader
  - readLine() - načíta riadok
  - read() - načíta znak
- FileReader extends InputStreamReader
  - new FileReader(file)
- InputStreamReader
  - new InputStreamReader(System.in)

Serializable

- Rozhranie - umožnuje "zabaliť" objekt
- Effective java – prefer alternatives to serializable
- serialVersionUID, Serializácia a deserializácia

```
FileOutputStream file = new FileOutputStream(filename);
ObjectOutputStream out = new ObjectOutputStream(file);
out.writeObject(object);
out.close(); file.close();

FileInputStream file = new FileInputStream(filename);
ObjectInputStream in = new ObjectInputStream(file);
Book object1 = (Book) in.readObject();
in.close(); file.close();
```

binárne súbory, nie textové

JavaDoc

- 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
- JavaDoc komentáre využívajú generátory dokumentácie a vývojové prostredia (IDE).
- Vygenerovanie v IntelliJ: Tools->Generate JavaDoc...

Maven

- Komplexný nástroj pre správu, riadenie a automatizáciou „buildov“ aplikacií
- Štandard vo svete Javy

- **artefakt** (artifact) – **základný prvok Mavenu -** niečo, čo je výsledkom projektu alebo je to použité projektom
- **archetyp** (archetype) – artefakt s predpripravenou šablónou projektu

JPAZ2 archetypy

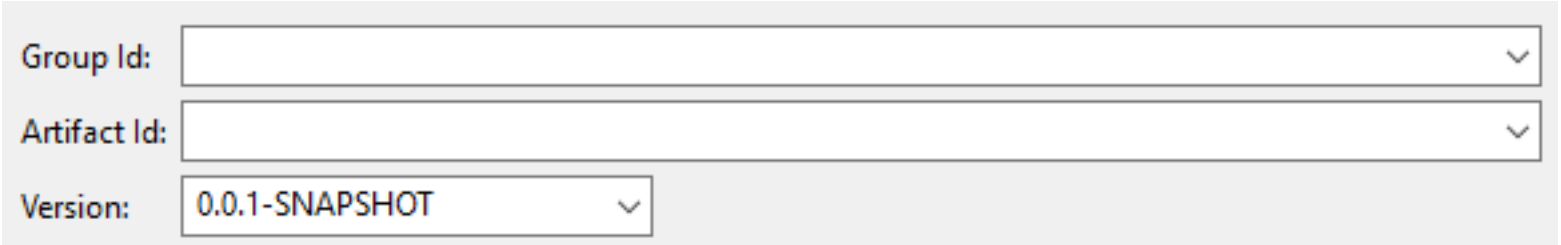
- Katalóg archetypov pre predmet PAZ1a:

  http://jpaz2.ics.upjs.sk/maven/archetype-catalog.xml

  - **jpaz2-archetype-novice**
  - **jpaz2-archetype-quickstart**
  - **jpaz2-archetype-launcher**
  - **jpaz2-archetype-theater**

Maven - artefakty

- Mavenovský projekt = Maven artefakt
- Identifikácia artefaktov:
  - **groupId** – jedinečná identifikácia skupiny artefaktov
  - **artifactId** – identifikácia artefaktu v rámci skupiny
    - pre nás: názov projektu
  - ~~**version** – verzia artefaktu~~
  - ~~**packaging** – typ výstupu~~

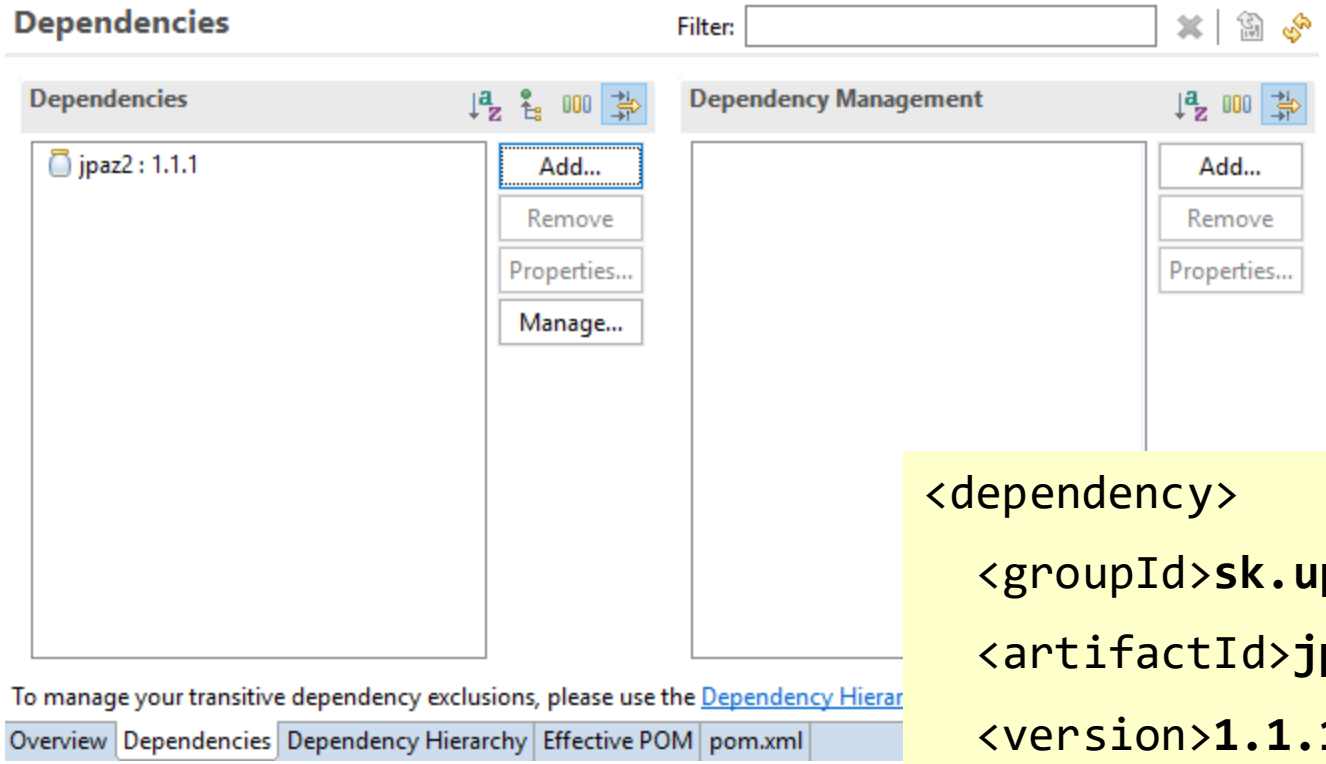


Maven - závislosti

- Chcem použiť nejakú „cool“ knižnicu vo svojom projekte (svojom artefakte) = pridám „cool“ knižnicu (jej artefakt) ako **závislosť** (dependency) do projektu
- **Závislosti** = iné artefakty, ktoré potrebujem pre fungovanie môjho projektu (artefaktu)
- **Zdroje artefaktov**:
  - Maven Central Repository
    - vyhľadávanie: https://search.maven.org/
  - vlastné (privátne) repozitáre
  - artefakty nainštalované v lokálnom repozitári (cache)

Takmer všetky známe i menej známe Java projekty.

Pridanie závislosti



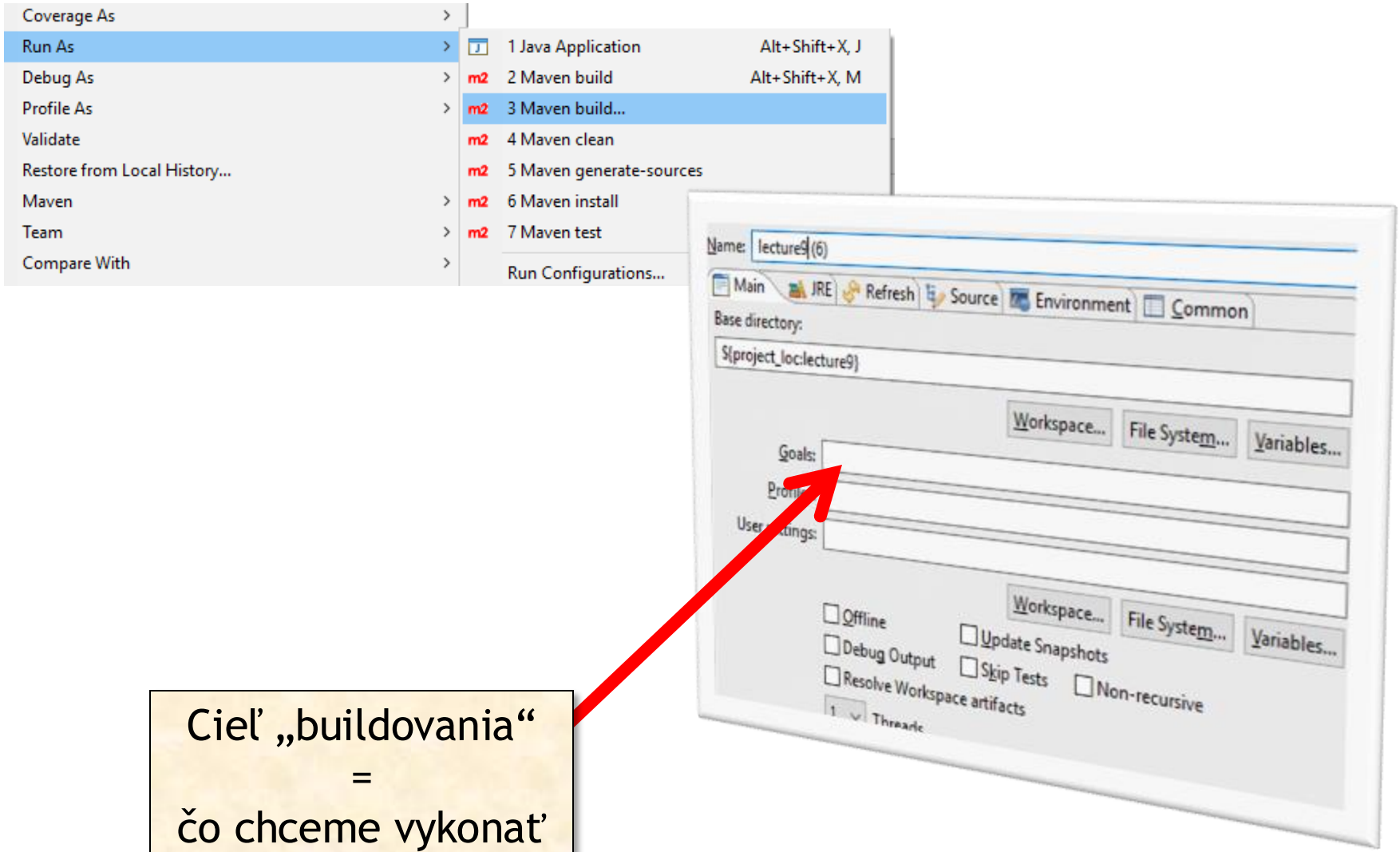

```
<dependency>
  <groupId>sk.upjs</groupId>
  <artifactId>jpaz2</artifactId>
  <version>1.1.1</version>
</dependency>
```

`pom.xml` – kompletný popis projektu

pom.xml

- pom.xml
  - Project Object Model
  - v xml formáte popisuje projekt
    - závislosti
    - pluginy
    - konfigurácie
    - proces zostavovania („buildovania“)
- Effective POM – reálne použitý pom.xml
  - Viac v Maven ponuke v IntelliJ IDEA

Buildovanie



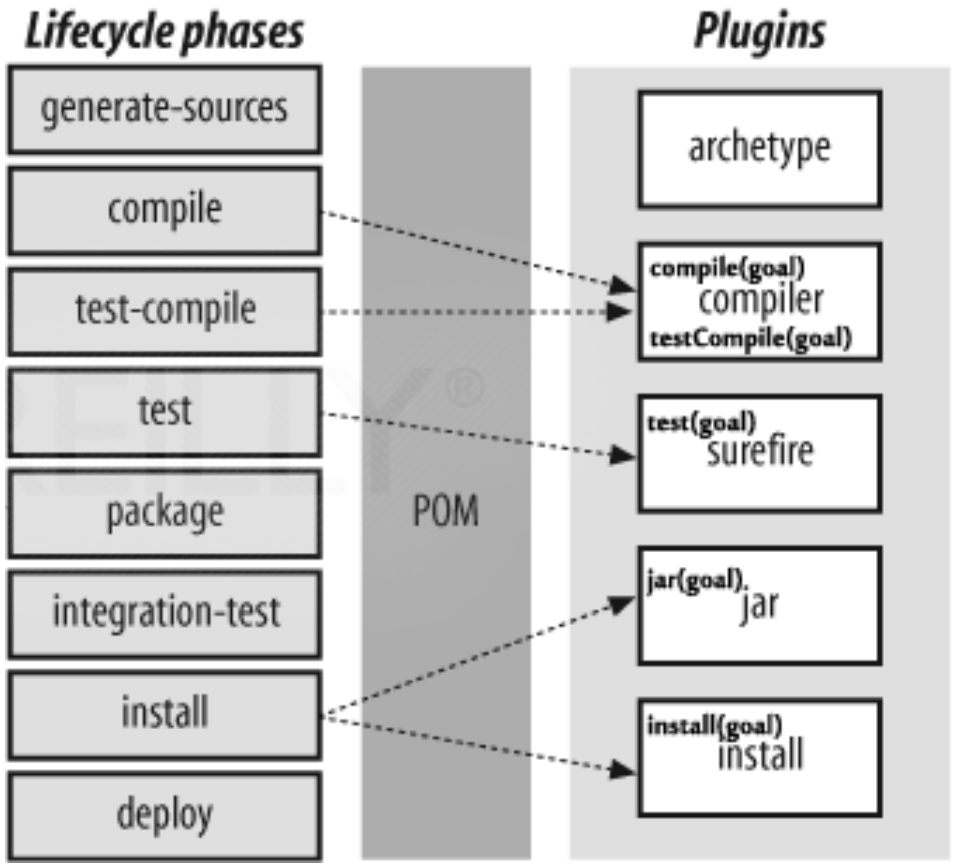
Cieľ „buildovania“
=
čo chceme vykonať

Ciele buildovania

- package
  - vytvorí jar-ko
    - ak je packaging nastavený na jar (iné zatiaľ nepoznáme)
  - vytvorí spustiteľné a minimalizované jar-ko
    - ak sa použije konfigurácia `pom.xml` akú vytvárajú napr. JPAZ2 archetypy
    - `<exec.mainClass>`*`trieda spúšťača`*`</exec.mainClass>`
  - výstup: v podadresári `target` projektu
- javadoc:javadoc
  - podľa javadoc komentárov vytvorí dokumentáciu
  - výstup: v podadresári `target/site/apidocs`

Ako to funguje?

Lifecycle = postupnosť **fáz buildovania**



Pluginy = vykonávajú **reálnu prácu** cez žiadosť na vykonanie nejakého cieľa

Ciele sa môžu „zaregistrovať“ do nejakých fáz (cez pom.xml alebo samé)

Čo napísať do goals?

- Goals:
  - fáza lifecycle = vykonaj **všetko po** danú fázu (vrátane)
    - napr. `package`
  - cieľ pluginu = vykonávaj life-cycle **až kým** nevykonáš cieľ
    - ak cieľ nie je „napojený“ na žiadnu fázu, len vykonaj cieľ
    - `javadoc:javadoc` = vykonaj cieľ `javadoc` pluginu `javadoc`.

Typické ciele

- `clean`
  - vyčisti (vymaž) všetky výstupy projektu
- `compile`
  - skompiluje projekt
- `site`
  - vytvorí dokumentáciu ku artefaktu
- `install`
  - package + ďalšie veci + inštalácia artefaktu do lokálneho repozitára

Návrat na úplný začiatok

- `public static void main(String[] args) { ... }`
- Spustenie
  - v IntelliJ IDEA
  - cez príkazový riadok
  - "dvojklikom" na jar súbor
- `java -jar nazovProjektu.jar`
- Argumenty:
  - Edit Configurations ... -> Program Arguments
  - `java -jar kopirovacCisel.jar cisla.txt vystup.txt ...`

Ďakujem za pozornosť !